conference

proceedings

HotOS XI: 11th Workshop on Hot Topics in Operating Systems

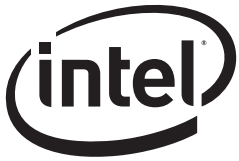
San Diego, CA, USA
May 7–9, 2007

Sponsored by
The USENIX Association

USENIX

in cooperation with the IEEE Technical Committee on Operating Systems (TCOS)

Thanks to Our Sponsors



USENIX Association

Proceedings of HotOS XI: 11th Workshop on Hot Topics in Operating Systems

May 7–9, 2007
San Diego, CA, USA

Workshop Organizers

Program Chairs

Galen Hunt, *Microsoft Research*

Program Committee

George Candea, *EPFL*
Landon Cox, *Duke University*
Armando Fox, *University of California, Berkeley*
Rebecca Isaacs, *Microsoft Research Cambridge*
Rodrigo Rodrigues, *Instituto Superior Técnico and INESC-ID*
Margo Seltzer, *Harvard University*
Michael Swift, *University of Wisconsin, Madison*
Amin Vahdat, *University of California, San Diego*
David Wetherall, *Intel Research and University of Washington*
John Wilkes, *Hewlett-Packard Labs*
Emmett Witchel, *University of Texas at Austin*
Yuanyuan Zhou, *University of Illinois at Urbana-Champaign*

Sponsorship Chair

Michael Swift, *University of Wisconsin, Madison*

Steering Committee

Michael B. Jones, *Microsoft, Inc.*
Margo Seltzer, *Harvard University*
Ellie Young, *USENIX Association*

The USENIX Association Staff

External Reviewers

James Anderson
Katerina Argyraki
Joao P. Barreto
Peter Bodik
Joao Cachopo
Emmanuel Cecchet
Olivier Crameri
Steve Dropsho
Daniel Ellard
Sameh Elniketi
Sasha Fedorova
Jason Flinn
Rodrigo Fonseca
Archana Ganapathi
Joao Garcia
Diwaker Gupta
Tim Harris
Val Henson
Owen S. Hofmann
Horatiu Jula
Charles Killian
Nikola Knezevic
Aravind Menon
Onur Mutlu
Donald E. Porter
George Porter
Hany E. Ramadan
Carlos Ribeiro
Christopher J. Rossbach
Nuno Santos
Scott Shenker
Jeff Shneidman
Joao Silva
Luis Veiga
Wei Xu
Willy Zwaenepoel

HotOS XI: 11th Workshop on Hot Topics in Operating Systems
May 7–9, 2007
San Diego, CA, USA

Monday, May 7

Coping with Concurrency

Is the Optimism in Optimistic Concurrency Warranted? 1
Donald E. Porter, Owen S. Hofmann, and Emmett Witchel, The University of Texas at Austin

Thread Scheduling for Multi-Core Platforms 7
Mohan Rajagopalan, Brian T. Lewis, and Todd A. Anderson, Programming Systems Lab, Intel

Automatic Mutual Exclusion 13
Michael Isard and Andrew Birrell, Microsoft Research, Silicon Valley

Modern Abstractions

Hype and Virtue 19
Timothy Roscoe, ETH Zürich; Kevin Elphinstone and Gernot Heiser, National ICT Australia

Relaxed Determinism: Making Redundant Execution on Multiprocessors Practical 25
Jesse Pool, Ian Sin Kwok Wong, and David Lie, University of Toronto

Compatibility Is Not Transparency: VMM Detection Myths and Realities 31
Tal Garfinkel, Stanford University; Keith Adams, VMware; Andrew Warfield, University of British Columbia/XenSource; Jason Franklin, Carnegie Mellon University

Algorithms for Profit

Don't Settle for Less Than the Best: Use Optimization to Make Decisions 37
Kimberly Keeton, Terence Kelly, Arif Merchant, Cipriano Santos, Janet Wiener, and Xiaoyun Zhu, Hewlett-Packard Laboratories; Dirk Beyer, M-Factor

Hyperspaces for Object Clustering and Approximate Matching in Peer-to-Peer Overlays 43
Bernard Wong, Ýmir Vigfússon, and Emin Gün Sirer, Cornell University

Optimizing Power Consumption in Large Scale Storage Systems 49
Lakshmi Ganesh, Hakim Weatherspoon, Mahesh Balakrishnan, and Ken Birman, Cornell University

Tuesday, May 8

Guarantees for the Future

Can Ferris Bueller Still Have His Day Off? Protecting Privacy in the Wireless Era 55
Ben Greenstein, Intel Research Seattle; Ramakrishna Gummadi, University of Southern California; Jeffrey Pang, Carnegie Mellon University; Mike Y. Chen, Intel Research Seattle; Tadayoshi Kohno, University of Washington; Srinivasan Seshan, Carnegie Mellon University; David Wetherall, University of Washington and Intel Research Seattle

Auditing to Keep Online Storage Services Honest 61
Mehul A. Shah, Mary Baker, Jeffrey C. Mogul, and Ram Swaminathan, HP Labs

A Web Based Covert File System 67
Arati Baliga, Joe Kilian, and Liviu Iftode, Rutgers University

Tuesday, May 8 (continued)

New Solutions, Old Problems

Purely Functional System Configuration Management 73
Eelco Dolstra, Utrecht University; Armijn Hemel, Loohuis Consulting

Processor Hardware Counter Statistics as a First-Class System Resource 79
Xiao Zhang, Sandhya Dwarkadas, Girts Folkmanis, and Kai Shen, University of Rochester

Microdrivers: A New Architecture for Device Drivers 85
Vinod Ganapathy, Arini Balakrishnan, Michael M. Swift, and Somesh Jha, University of Wisconsin—Madison

Wednesday, May 9

Web 2.0

MashupOS: Operating System Abstractions for Client Mashups 91
Jon Howell, Microsoft Research; Collin Jackson, Stanford University; Helen J. Wang and Xiaofeng Fan, Microsoft Research

Live Monitoring: Using Adaptive Instrumentation and Analysis to Debug and Maintain Web Applications 99
Emre Kıcıman and Helen J. Wang, Microsoft Research

End-to-End Web Application Security 105
Úlfar Erlingsson, Benjamin Livshits, and Yinglian Xie, Microsoft Research

Finding a Better Way

HotComments: How to Make Program Comments More Useful? 111
Lin Tan, Ding Yuan, and Yuanyuan Zhou, University of Illinois at Urbana-Champaign

Towards a Practical, Verified Kernel 117
Kevin Elphinstone and Gerwin Klein, National ICT Australia and the University of New South Wales; Philip Derrin, National ICT Australia; Timothy Roscoe, ETH Zürich; Gernot Heiser, National ICT Australia, the University of New South Wales, and Open Kernel Labs

Beyond Bug-Finding: Sound Program Analysis for Linux 123
Zachary Anderson, Eric Brewer, and Jeremy Condit, University of California, Berkeley; Robert Ennals and David Gay, Intel Research Berkeley; Matthew Harren, George C. Necula, and Feng Zhou, University of California, Berkeley

Index of Authors

Adams, Keith 31
Anderson, Todd A. 7
Anderson, Zachary 123
Baker, Mary 61
Balakrishnan, Arini 85
Balakrishnan, Mahesh 49
Baliga, Arati 67
Beyer, Dirk 37
Birman, Ken 49
Birrell, Andrew 13
Brewer, Eric 123
Chen, Mike Y. 55
Condit, Jeremy 123
Derrin, Philip 117
Dolstra, Eelco 73
Dwarkadas, Sandhya 79
Elphinstone, Kevin 19, 117
Ennals, Robert 123
Erlingsson, Úlfar 105
Fan, Xiaofeng 91
Folkmanis, Girts 79
Franklin, Jason 31
Ganapathy, Vinod 85
Ganesh, Lakshmi 49
Garfinkel, Tal 31
Gay, David 123
Greenstein, Ben 55
Gummadi, Ramakrishna 55
Harren, Matthew 123
Heiser, Gernot 19, 117
Hemel, Armijn 73
Hofman, Owen S. 1
Howell, Jon 91
Iftode, Liviu 67
Isard, Michael 13
Jackson, Collin 91
Jha, Somesh 85
Keeton, Kimberly 37
Kelly, Terence 37
Kıcıman, Emre 99
Kilian, Joe 67
Klein, Gerwin 117
Kohno, Tadayoshi 55
Lewis, Brian T. 7
Lie, David 25
Livshits, Benjamin 105
Merchant, Arif 37
Mogul, Jeffrey C. 61
Necula, George C. 123
Pang, Jeffrey 55
Pool, Jesse 25
Porter, Donald E. 1
Rajagopalan, Mohan 7
Roscoe, Timothy 19, 117
Santos, Cipriano 37
Seshan, Srinivasan 55
Shah, Mehul A. 61
Shen, Kai 79
Sirer, Emin Gün 43
Swaminathan, Ram 61
Swift, Michael M. 85
Tan, Lin 111
Vigfússon, Ýmir 43
Wang, Helen J. 91, 99
Warfield, Andrew 31
Weatherspoon, Hakim 49
Wetherall, David 55
Wiener, Janet 37
Witchel, Emmett 1
Wong, Bernard 43
Wong, Ian Sin Kwok 25
Xie, Yinglian 105
Yuan, Ding 111
Zhang, Xiao 79
Zhou, Feng 123
Zhou, Yuanyuan 111
Zhu, Xiaoyun 37

Is the Optimism in Optimistic Concurrency Warranted?

Donald E. Porter, Owen S. Hofmann, and Emmett Witchel
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712
{porterde, osh, witchel}@cs.utexas.edu

Abstract

Optimistic synchronization allows concurrent execution of critical sections while performing dynamic conflict detection and recovery. Optimistic synchronization will increase performance only if critical regions are *data independent*—concurrent critical sections access disjoint data most of the time. Optimistic synchronization primitives, such as transactional memory, will improve the performance of complex systems like an operating system kernel only if the kernel's critical regions have reasonably high rates of data independence.

This paper introduces a novel method and a tool called *syncchar* for exploring the potential benefit of optimistic synchronization by measuring data independence of potentially concurrent critical sections. Experimental data indicate that the Linux kernel has enough data independent critical sections to benefit from optimistic concurrency on smaller multiprocessors. Achieving further scalability will require data structure reorganization to increase data independence.

1 Introduction

As CPU manufacturers have shifted from scaling clock frequency to placing multiple simple processor cores on one chip, there has been a renewed interest in concurrent programming. The end-user benefit of these systems will be limited unless software developers can effectively leverage the parallel hardware provided by these new processors.

Concurrent programming in a shared memory system requires primitives such as locks to synchronize threads of execution. Locks have many known problems, including deadlock, convoying, and priority inversion that make concurrent programming in a shared memory model difficult. In addition, locks are a *conservative* synchronization primitive—they always assure mutual exclusion, regardless of whether threads actually need to execute a critical section sequentially for correctness. Consider modifying elements in a binary search tree. If the tree is large and the modifications are evenly distributed, most modifications can safely occur in parallel. A lock protecting the entire tree will needlessly serialize modifications.

One solution for the problem of conservative locking is to synchronize data accesses at a finer granularity–rather than lock an entire binary search tree, lock only the individual nodes being modified. This presents two problems. First, data structure invariants enforce a lower bound on the locking granularity. In some data structures, this bound may be too high to fully realize the available data parallelism. Second, breaking coarse-grained locks into many fine-grained locks significantly increases code complexity. As the locking scheme becomes more complicated, long-term correctness and maintainability are jeopardized.

An alternative to conservative locking is *optimistic concurrency*. In an optimistic system, concurrent accesses to shared data are allowed to proceed, dynamically detecting and recovering from conflicting accesses. A specialized form of optimistic concurrency is lock-free data structures (including many variants like wait-free and obstruction-free data structures) [4, 5]. Lock-free data structures, while optimistic, are not a general purpose solution. Lock-free data structures require that each data structure's implementation meets certain non-trivial correctness conditions. There is also no general method to atomically move data among different lock-free data structures.

Transactional memory [6] provides hardware or software support for designating arbitrary regions of code to appear to execute with atomicity, isolation and consistency. Transactions provide a generic mechanism for optimistic concurrency by allowing critical sections to execute concurrently and automatically roll-back their effects on a data conflict. Coarse-grained transactions are able to reduce code complexity while retaining the concurrency of fine-grained locks.

To benefit from optimistic concurrency, however, critical sections must have a substantial amount of *data independence*—data written by one critical section must be disjoint from data read or written by critical sections of concurrently executing threads. If critical sections concurrently modify the same data, or have *data con-*

Critical Section 1	Critical Section 2	Critical Section 3
begin critical section; node = root→right; node→left = root→left→right; end critical section;	begin critical section; node = root→left; node→left = root→left→right; end critical section;	begin critical section; node = root; node→right = node→left; end critical section;
r: 0x1000, 0x1032*, 0x1064, 0x2000 w: 0x2064, 0x3000, 0x3064, 0x3032	r: 0x1000, 0x1032*, 0x1064, 0x2000 w: 0x2064, 0x3000, 0x3064, 0x2032	r: 0x1000, 0x1064 w: 0x1032*

Table 1: Three critical sections that could execute on the tree in Figure 1 and their address sets. The read entries marked with an asterisk (*) are conflicting with the write in Critical Section 3.

flicts[1], optimistic concurrency control will serialize access to critical sections. In this case optimistic control can perform much worse than conservative locking due to the overhead required to detect and resolve conflicts.

In this paper, we present novel techniques (Section 2) and a tool (Section 3) for investigating the limits of optimistic concurrency by measuring the data independence of critical regions that would be protected by the same lock. We apply the methodology and the tool to the Linux kernel, and present results (Section 4).

2 The Limits of Optimistic Concurrency

The most general way to determine data independence is to compare the *address sets* of the critical sections. The address set of a critical section is the set of memory locations read or written. If the code in critical sections access disjoint memory locations, the effect of executing them concurrently will be the same as executing them serially. In addition, the address sets need not be entirely disjoint; only the write set of each critical section must be disjoint from the address set of other potentially concurrent critical sections. In other words, it is harmless for multiple critical sections to read the same data as long as that data is not concurrently written by another. This criterion is known as conflict serializability in databases, and it is widely used due to the relative ease of detecting a conflict.

Conflict serializability is a pessimistic model, however, as two critical sections can conflict yet be safely executed in parallel. For example, if two critical sections conflict only on their final write, they can still safely execute concurrently if one finishes before the other issues the conflicting write. Some data structures and algorithms make stronger guarantees that allow critical sections to write concurrently to the same locations safely, but these are relatively rare and beyond the scope of this paper. Such guarantees correspond to view serializability; computing view serializability is NP-complete and hence rarely used in practice [9]. This paper will use conflict serializability exclusively.

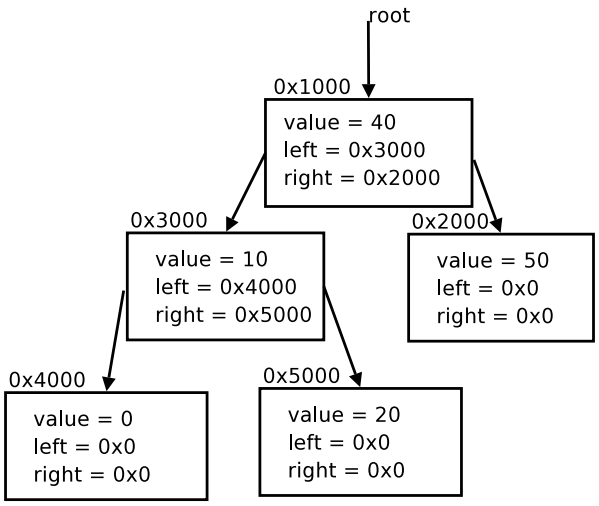


Figure 1: A simple binary tree.

As an example of conflict serializability, consider the simple binary tree in Figure 1. Three different critical sections that operate on this tree, along with their address sets are listed in Table 1. Critical sections 1 and 2 read many of the same memory locations, but only write to locations that are not in the other's address sets. They are, therefore, data independent and could safely execute concurrently. This makes sense intuitively because they modify different branches of the tree. Critical section 3, however, modifies the right pointer in the root of the tree, which cannot execute concurrently with critical sections that operate on the right branch of the tree. This is reflected in the address sets: 0x1032 is in Critical Section 3's write set and in the read sets of Critical Sections 1 and 2.

In our simple example, we can determine the data in-

[1]We selected the term data conflicts over data dependence to avoid confusion with other meanings.

dependence of the critical sections by simple static analysis. In most cases, however, static analysis is insufficient because functions that modify a data structure generally determine what to modify based on an input parameter. Thus, the data independence of critical section executions largely depends on the program's input, requiring investigation of the common case synchronization behavior in order to determine whether to employ optimistic concurrency.

There are cases where the structure of the critical section code can limit concurrency. If most executions of a critical section are not data independent, but only conflict on a small portion of the address set, finer-grained locking or data structure reorganization may remove these conflicts. For instance, the SLOB allocator in the Linux kernel uses a shared pointer to available heap space that is read at the beginning and written at the end of every allocation. Thus, any two allocations will have needless conflicts on this pointer and will never be data independent.

There are also cases where a high degree of data independence can be an argument for consolidation of overlapping critical sections under optimistic concurrency. In a program with fine-grained locking, multiple lock acquires and releases can be nested during overlapping critical sections to minimize the time a lock is held and increase performance. If, in an optimistic model, the outermost critical section is generally data independent, avoiding nested critical sections yields a worthwhile reduction in code complexity and potential increase in performance.

3 Syncchar

To measure the data independence of critical sections, we present a tool called *syncchar* (synchronization characterization) that runs as a module in Virtutech Simics, version 3.0.17 [7]. Simics is a full-system, execution-based simulator. Syncchar tracks the synchronization behavior of the Linux kernel, including each lock acquire and release, contention for a lock acquire, and tracks the address set of memory locations read and written in the critical section.

If a thread busy-waits to acquire a lock in the kernel, syncchar compares the thread's address set when it completes the critical section to the address sets of all critical sections it waited on to determine if they conflict. If two threads that waited for the same lock touch completely disjoint data, then they both could have optimistically executed the critical section concurrently, rather than needlessly waiting on a lock.

Comparing the address set of threads that happen to contend for a lock during one execution provides only a limited window into the latent concurrency of a lock-based application. To determine a more general limit on optimistic concurrency, when a lock is released, syncchar compares its address set to the address sets of the previous 128 critical sections protected by that lock. By recording and comparing the address sets for multiple critical sections, syncchar's measurements are less sensitive to the particular thread interleaving of the measured execution.

An ideal analysis would identify and compare all possible concurrent executions, rather than just the last 128. However, this would require tracking events such as thread forks and joins that structure concurrent execution. In addition, the number of possibly concurrent executions could make comparing address sets infeasible for applications of substantial length. Comparing over a window of critical section executions captures many executions that are likely to be concurrent, while minimizing false conflicts that result from serialization through other mechanisms.

Syncchar supports a window of 128 critical sections based on a sensitivity study of the window size. Changing the window size from 50 to 100 affects the data independence results for most locks by less than 5%. As the largest commercial CMP effort to date is an 80 core machine [1], 128 also represents a reasonable upper bound on the size of CMP's likely to be developed in the near future.

Syncchar only compares critical regions across different threads. Determining whether two executions of a critical section in the same thread can be parallelized is a more complex problem than determining whether critical sections from different threads can be executed concurrently. If possible, this would require substantial code rewriting or some sort of thread-level speculation [14]. However, this paper focuses only on the benefits of replacing existing critical sections with transactions.

Syncchar filters a few types of memory accesses from the address sets of critical sections to avoid false conflicts. First, it filters out addresses of lock variables, which are by necessity modified in every critical section. It filters all lock addresses, not just the current lock address, because multiple locks can be held simultaneously and because optimistic synchronization eliminates reads and writes of lock variables. Syncchar also filters stack addresses to avoid conflicts due to reuse of the same stack address in different activation frames.

Some kernel objects, like directory cache entries, are recycled through a cache. The lifetime of locks contained in such objects is bounded by the lifetime of the object itself. When a directory cache entry emerges from the free pool, its lock is initialized and syncchar considers it a new, active lock. The lock is considered inactive when the object is released back to the free pool. If the lock is made active again, its the address set history is

Lock	Total Acquires	Contended Acq	Data Conflicts	Mean Addr Set Bytes	Mean Confl Bytes
zone_lock	14,450	396 (2.74%)	100.00%	161	50
ide_lock	4,669	212 (4.54%)	97.17%	258	24
runqueue_t.lock (0xc1807500)	4,616	143 (3.23%)	84.62%	780	112
zone.lru_lock	16,186	131 (0.81%)	48.09%	134	8
rcu_ctrlblk.lock	10,975	84 (0.77%)	97.62%	27	4
inode.i_data.i_mmap_lock	1,953	69 (3.53%)	89.86%	343	48
runqueue_t.lock (0xc180f500)	3,686	62 (1.74%)	90.32%	745	118
runqueue_t.lock (0xc1847500)	2,814	27 (0.96%)	88.89%	530	74
runqueue_t.lock (0xc1837500)	2,987	24 (0.80%)	95.83%	526	100
runqueue_t.lock (0xc184f500)	3,523	22 (0.68%)	86.36%	416	70
runqueue_t.lock (0xc182f500)	2,902	24 (0.83%)	87.50%	817	103
runqueue_t.lock (0xc1817500)	3,224	17 (0.68%)	94.12%	772	108
runqueue_t.lock (0xc1857500)	2,433	20 (0.86%)	90.00%	624	100
dcache_lock	15,358	21 (0.14%)	0.00%	0	0
files_lock	7,334	20 (0.27%)	70.00%	15	8

Table 2: The fifteen most contended spin locks during the pmake benchmark. Total Acquires is the number of times the lock was acquired by a process. Different instances of the same lock are distinguished by their virtual address. Contended Acq is the number of acquires that required a process had to spin before obtaining the lock, including the percent of total acquires. The Data Conflicts column lists the percentage of Contended Acquires that had a data conflict. Mean Addr Set Bytes is the average address set size of conflicting critical sections, whereas Mean Confl Bytes is the average number of conflicting bytes.

cleared, even though it resides at the same address as in its previous incarnation.

Before scheduling a new process, the kernel acquires a lock protecting one of the runqueues. This lock is held across the context switch and released by a different process. As this lock is actually held by two processes during one critical section, Syncchar avoids comparing its address set to prior address sets from either process.

Finally, some spinlocks protect against concurrent attempts to execute I/O operations, but do not actually conflict on non-I/O memory addresses. Syncchar cannot currently detect I/O, so results for locks that serialize I/O may falsely appear data independent.

4 Experimental Results

We run our experiments on a simulated machine with 15 1 GHz Pentium 4 CPUs and 1 GB RAM. We use 15 CPUs because that is the maximum supported by the Intel 440-bx chipset simulated by Simics. For simplicity, our simulations have an IPC of 1 and a fixed disk access latency of 5.5ms. Each processor has a 16KB instruction cache and a 16KB data cache with a 0 cycle access latency. There is a shared 4MB L2 cache with an access time of 16 cycles. Main memory has an access time of 200 cycles.

We use version 2.6.16.1 of the Linux kernel. To simulate a realistic software development environment, our experiments run the pmake benchmark, which executes `make -j 30` to compile 27 source files from the libFLAC 1.1.2 source tree in parallel.

4.1 Data Independence of Contended Locks

The data independence of the most contended kernel locks during the pmake benchmark is presented in Table 2. Columns 2 and 3 show the total number of times each lock was acquired and the percentage of those acquires which were contended. There are low levels of lock contention in the kernel for the pmake workload. For all locks, at least 95% of acquires are uncontended. Linux developers have invested heavy engineering effort to make kernel locks fine-grained.

Each time two or more threads contend for the same lock, syncchar determines whether the critical sections they executed have a data conflict (shown in column 4). These locks have a high rate of data conflict, indicating that 80–100% of the critical sections cannot be executed safely in parallel. Many of the locks in this list protect the process runqueues, which are linked lists. The linked list data structure is particularly ill-suited for optimistic concurrency because modification of an element near the front will conflict with all accesses of elements after it in the list.

There is one noteworthy lock that protects critical regions that are highly data independent—`dcache_lock`, a global, coarse-grained lock. The `dcache_lock` protects the data structures associated with the cache of directory entries for the virtual filesystem. Directory entries represent all types of files, enabling quick resolution of path names. This lock is held during a wide range of filesystem operations that often access disjoint regions of the directory entry cache.

In the cases where the critical sections have data con-

flicts, we measured the number of bytes in each address set and the number of conflicting bytes, listed in Columns 5 and 6. A particularly small address set can indicate that the lock is already fine grained and has little potential for optimistic execution. For example, the `rcu_ctrlblk.lock`, protects a small, global control structure used to manage work for a Read-copy update (RCU). When this lock is held, only a subset of the structure's 4 integers and per-CPU bitmap are modified and the lock is immediately released. Our experimental data closely follows this pattern, with the lock's critical sections accessing an average of 7 bytes, 5 of which conflict.

A larger working set with only a small set of conflicting bytes can indicate an opportunity for reorganizing the data structure to be more amenable to optimistic synchronization. The `zone.lru_lock` protects two linked lists of pages that are searched to identify page frames that can be reclaimed. Replacing the linked list with a data structure that avoided conflicts on traversal could substantially increase the level of data independence.

Lock	Confl Acq	Mean Confl Bytes	
zone.lock	100.00%	32.49	20.40%
zone.lru_lock	65.82%	6.63	20.10%
ide_lock	70.92%	6.93	19.03%
runqueue_t.lock (0xc1807500)	27.41%	24.06	24.50%
runqueue_t.lock (0xc180f500)	29.14%	26.10	23.20%
inode.i_data.i_mmap_lock	46.03%	22.81	16.53%
runqueue_t.lock (0xc1837500)	27.30%	27.83	19.15%
seqlock_t.lock	100.00%	56.01	48.41%
runqueue_t.lock (0xc1847500)	23.32%	26.64	19.55%
runqueue_t.lock (0xc183f500)	21.03%	28.21	18.37%

Table 3: Limit study data from the ten longest held kernel spin locks during the pmake benchmark. This data is taken from comparing each address set to the last 128 address sets for that lock, rather than contended acquires, as in Table 2. Conflicting Acquires are the percent of acquires that have conflicting data accesses. Mean Conflicting Bytes shows the average number of conflicting bytes and their percentage of the total address set size.

4.2 The Limits of Kernel Data Independence

To investigate the limits of data independence in the kernel, syncchar compares the address set of each critical section to the address sets of the last 128 address sets for the same lock, as discussed in Section 3. The purpose of comparing against multiple address sets is to investigate how much inherent concurrency is present in the kernel.

Amdahl's law governs the speedup gained by exploiting the concurrency inherent within Linux's critical sections. Locks that are held for the longest period of time contribute the most to any parallel speedup, so the results in Table 3 are presented for the ten longest held locks. Conflicting acquires vary from 21%–100%, though the average data independence across all spinlocks, weighted by the number of cycles they are held, is 24.1%. That level of data independence will keep an average of 4 processors busy. Data structure reorganization can increase the amount of data independence.

One interesting lock in Table 3 is the `seqlock_t.lock`. Seqlocks are a kernel synchronization primitive that provides a form of optimistic concurrency by allowing readers to speculatively read a data structure. Readers detect concurrent modification by checking a sequence number before and after reading. If the data structure was modified, the readers retry. Seqlocks use a spinlock internally to protect against concurrent increments of the sequence number by writers, effectively serializing writes to the protected data. Because the same sequence number is modified every time the lock is held, this lock's critical section will never be data independent. Although this seems to limit concurrency at first blush, there remains a great deal of parallelism in the construct because an arbitrarily large number of processes can read the data protected by the seqlock in parallel. Sometimes locks that protect conflicting data sets are not indicators of limited concurrency because they enable concurrency at a higher level of abstraction.

The data conflicts of fine-grained locks can distract attention from the ability to concurrently access the larger data structure. The locks protecting the per-CPU data structure `tvec_base_t` from an occasional access by another CPU tend to have an extremely low degree of data independence because they are fine grained. In the common case, however, the data these locks protect is only accessed by one CPU, which represents a large degree of overall concurrency. We leave extending this methodology to multiple levels of abstraction for future work.

4.3 Transactional Memory

To provide a comparison between the performance of lock-based synchronization and optimistic synchronization, we ran the pmake benchmark under syncchar on both Linux and on TxLinux with the MetaTM hardware transactional memory model [13]. TxLinux is a version of Linux that has several key synchronization primitives converted to use hardware transactions. Syncchar measures the time spent acquiring spinlocks and the time lost to restarted transactions. For this workload, TxLinux converts 32% of the spinlock acquires in Linux to transactions, reducing the time lost to synchronization overhead by 8%. This reduction is largely attributable to removing cache misses for the spinlock lock variable. The

syncchar data indicates that converting more Linux locks to transactions will yield speedups that can be exploited by 4–8 processors. However, gaining even greater concurrency requires data structure redesign.

5 Related work

This paper employs techniques from parallel programming tools to evaluate the limits of optimistic concurrency. There is a large body of previous work on debugging and performance tuning tools for programs [3, 15]. Our work is different from other tools because it is concerned more with identifying upper bounds on performance rather than performance tuning or verifying correctness.

Lock-free (and modern variants like obstruction-free) data structures are data-structure specific approaches to optimistic concurrency [4, 5]. Lock-free data structures attempt to change a data structure optimistically, dynamically detecting and recovering from conflicting accesses.

Transactional memory is a more general form of optimistic concurrency that allows arbitrary code to be executed atomically. Herlihy and Moss [6] introduced one of the earliest Transactional Memory systems. More recently, Speculative Lock Elision [10] and Transactional Lock Removal [11] optimistically execute lock regions transactionally. Several designs for full-function transactional memory hardware have been proposed [2,8,12].

6 Conclusion

This paper introduces a new methodology and tool for examining the potential benefits of optimistic concurrency, based on measuring the data independence of critical section executions. Early results indicate that there is sufficient data independence to warrant the use of optimistic concurrency on smaller multiprocessors, but data structure reorganization will be necessary to realize greater scalability. The paper motivates more detailed study of the relationship of concurrency enabled at different levels of abstraction, and the ease and efficacy of data structure redesign.

7 Acknowledgements

We would like to thank the anonymous reviewers, Hany Ramadan, Christopher Rossbach, and Virtutech AB. This research is supported in part by NSF CISE Research Infrastructure Grant EIA-0303609 and NSF Career Award 0644205.

References

[1] Michael Feldman. Getting serious about transactional memory. *HPCWire*, 2007.

[2] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[3] J. Harrow. Runtime checking of multithreaded applications with visual threads. In *SPIN*, pages 331–342, 2000.

[4] M. Herlihy. Wait-free synchronization. In *TOPLAS*, January 1991.

[5] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, 2003.

[6] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.

[7] P.S. Magnusson, M. Christianson, and J. Eskilson et al. Simics: A full system simulation platform. In *IEEE Computer vol.35 no.2*, Feb 2002.

[8] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA*, 2006.

[9] C. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[10] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, 2001.

[11] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, October 2002.

[12] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA*. 2005.

[13] H.E. Ramadan, C.J. Rossbach, D.E. Porter, O.S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *ISCA*, 2007.

[14] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, 2000.

[15] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

Thread Scheduling for Multi-Core Platforms

Mohan Rajagopalan Brian T. Lewis Todd A. Anderson
Programming Systems Lab, Intel Corporation
Santa Clara, CA 94054
{*mohan.rajagopalan, brian.t.lewis, todd.a.anderson*}*@intel.com*

Abstract

As multi-core processors with tens or hundreds of cores begin to proliferate, system optimization issues once faced only by the high-performance computing (HPC) community will become important to all programmers. However, unlike with HPC, the focus of the multi-core programmer will be on programming productivity and portability as much as performance. We introduce in this paper a novel scheduling framework for multi-core processors that strikes a balance between control over the system and the level of abstraction. Our framework uses high-level information supplied by the user to guide thread scheduling and also, where necessary, gives the programmer fine control over thread placement.

1 Introduction

Single-threaded processor performance is becoming power limited, so processor architects are increasingly turning to multi-core designs to improve processor performance. Such chips include Intel's experimental TeraFlops processor [9] and Sun's UltraSPARC T1 processor [10]. This design trend points to future systems having tens to hundreds of cores per processor, each of which is capable of running one or more software threads simultaneously. Experiments show that a wide variety of programs can benefit from the hardware parallelism [14] that such *many-core processors* (large-scale multithreaded chip multiprocessors) will provide.

Many-core processors will bring to desktop computing power formerly seen only in HPC systems. HPC programmers have traditionally used explicit thread and data placement to achieve the best performance for their parallel applications—an approach that not only requires a deep understanding of the hardware architecture of the HPC system, but also requires careful tailoring of the program to that specific hardware. We expect the number of people writing parallel programs to increase significantly. However, portability and ease of programming are equally, if not more, important to the general programmer than performance. Since the architecture of many-core systems is still evolving, portability is needed to allow the same program to run well on different kinds of many-core systems. As a result, one challenge for mainstream many-core programming is to develop mechanisms that provide the ability to do HPC-style customization for performance but do not compromise portability and programmability.

There are a number of challenges involved in designing a portable and easy-to-use, yet high performance scheduling interface for many-core platforms. For example, it will be essential to reduce shared cache misses since the on-die caches of many-core processors will be relatively small for at least the next several years, and memory latencies have grown to several hundred cycles [7]. Similarly, choosing which threads run concurrently on a processor is important since cache contention and bus traffic can significantly impact application performance. It can also be important to decide which threads to run on each core since simultaneous-multithreaded (SMT) cores share low-level hardware resources such as TLBs among all threads.

To achieve the best performance for their parallel applications, programmers have traditionally used explicit thread and data placement. Most thread library implementations provide support for *pinning threads* to assign threads to specific CPUs (i.e., hardware threads) and to restrict their migration [11, 6]. But doing this requires the programmer to have a deep understanding of the system's architecture, which significantly hampers program portability and programmability. Similar problems exist for language-based HPC approaches [3, 1, 4] that are based on specifying *locales* or *regions* for computation.

Another challenge is that the scheduling primitives must support a wide variety of parallelization requirements. Moreover, some applications need different scheduling strategies for different program phases. The threads of data-parallel applications, for example, are typically independent: they share only a small amount of data except at certain communication points. For these applications, distributing threads widely among the different CPUs is beneficial. On the other hand, the threads of array-based programs typically share data heavily, so scheduling the threads on nearby CPUs to share data in

common caches provides the best performance.

This paper introduces a scheduling framework for many-core systems that tries to strike a balance between the level of abstraction and the amount of control over the underlying system. This framework is based on the concept of the *Related Thread ID* (*RTID*), which is used to identify a collection of related software threads to the thread scheduler in order to improve their runtime performance. For example, a group of threads might be given a common RTID because they share some data or lock. RTIDs provide a higher level of abstraction than traditional fixed thread-to-processor mappings. In addition, RTIDs allow the programmer to express various scheduling constraints such as whether threads should be gang scheduled. We are currently finishing an initial implementation of our RTID-based scheduling framework as an extension of our existing McRT many-core runtime system [14]. This paper introduces RTIDs, presents the RTID interface our framework provides to user-level code, and describes our initial approach to the RTID-based scheduler design.

The remainder of the paper is organized as follows. The next section describes characteristics of the many-core architectures that we target. We present a detailed description of our framework and its interface API in Section 3, then Section 4 discusses how our scheduler framework addresses a variety of design issues. This is followed in Section 5 by an overview of our scheduler's implementation. Section 6 describes related work. The last section concludes and discusses future directions.

2 Target architecture

The underlying processor architecture plays a significant role in thread scheduling. While the exact architecture of large-scale many-core processors is still evolving, we can list some general characteristics.

Many-core processors will have tens to hundreds of cores, each of which run application threads on some number of hardware threads (HWTs). Initially, each HWT will have a private L1 cache and each core will have a shared L2 cache shared by all HWTs. Processors are likely to eventually have multiple levels of private cache per HWT, as well as a shared last-level cache. However, the many-core L2 caches will tend to be significantly smaller than those of traditional SMP systems. In addition, the cores will be interconnected by a high-bandwidth, low-latency interconnect.

The architectural differences of many-core processors have a number of consequences. First, on-die communication will be fast: the latency to access data from a different HWT will be about two orders of magnitude smaller than in today's SMP systems. Also, the cache size for each HWT and each core will be one or two orders of magnitude smaller than the per-processor caches of today's SMP systems. These differences mean that parallelism will be relatively cheap as long as the required data (and instructions) are on-die. However, the design of the interconnection fabric will also significantly affect latency, so thread placement decisions will have a big impact on thread performance.

In our current scheduling framework, we target single many-core processors. We expect to extend our framework to multiple many-core processors in the future. In addition, we also assume initially that the processor is homogeneous: that all cores are identical. However, future processors may be heterogeneous. For example, some cores may be faster, or may have special support for vector processing or other computation. We expect to later extend our framework to support these heterogeneous processors. Finally, we assume for now that the processor provides a coherent shared memory model with a single shared address space although this may have NUMA characteristics due to hierarchical caches. We do not specifically address distributed address spaces although we may do so in the future.

3 The scheduling framework

The primary goals of our scheduling framework are to improve application throughput and overall system utilization. A secondary goal of the framework is to improve fairness so that each thread continues to make good forward progress. These goals sometimes conflict. For example, to balance system load, the placement framework may schedule threads to run on HWTs distant from shared data. Also, scheduling threads to run too closely (for example, packed onto adjacent HWTs) can actually hurt performance if cache conflicts become too frequent. Our framework treats the question of how to balance these goals as a policy decision and lets the user specify which goal has the highest priority.

The framework provides support for both user-guided explicit placement as well as automatic best-effort placement. Expert programmers who want to control exactly where threads are scheduled can use explicit placement functions to control where to run threads and allocate data. However, we expect most users to use automatic placement since this simplifies application development and allows the same program to run well on a range of different many-core systems. In this scheme, the programmer identifies closely-related threads–threads that share data–and the runtime will do its best to schedule the threads close together so that they share data through nearby shared caches. In addition, the framework also provides a mechanism for *gang scheduling* threads—running them at the same time as well as close together. This feature is useful, for example, in programs where threads frequently acquire and release shared locks. In

this case, latency will suffer if threads made newly-runnable must wait until they can be scheduled to run on a CPU.

At the heart of our framework is a configurable scheduler that decides when and the HWT on which each thread will be run. Our scheduling algorithm is tailored for efficiently running one or more applications using automatic placement. This approach enables both performance and portability since architecture-specific optimizations are implemented within the scheduler. If the underlying architecture changes, the existing scheduler can be replaced with one that is tailored for the new architecture.

3.1 RTIDs

RTIDs provide a mechanism for programmers to identify groups of threads that should run close together, for example, because they share common data. As an example, RTIDs can be used to schedule threads so that they can share data through local caches. The scheduler tries its best to run threads that share an RTID on nearby HWTs or cores. In our current design, each thread can have at most one RTID. If a thread has an RTID, it is specified when the thread is created. Threads without RTIDs can be scheduled to run anywhere.

Our framework also allows a programmer to directly associate an RTID with a data address. The scheduler will then try to run threads with that RTID close to each other and to the specified data: that is, it will try to run the threads on HWTs that share fast caches in order to minimize cache misses.

Finally, the framework supports attributes for RTIDs that supply additional information to help guide how threads sharing that RTID are scheduled. The first attribute is the expected number of simultaneous threads sharing the RTID. The scheduler will use this information to select (at least initially) which HWTs to use for the RTID's threads. Two other attributes specify whether to gang schedule the RTID's threads, and, if so, the minimum number of threads in the gang. These values allow the scheduler to decide when a quorum of the RTID's threads is ready to run. A fourth attribute allows the programmer to specify whether load balancing is more important than data proximity. This can be used, for example, to help optimize scheduling for data-parallel algorithms, where threads are mostly independent and communicate infrequently. This attribute can also be used to help spread out threads sharing an RTID so as to avoid cache conflicts.

Because a program's requirements can change over its lifetime, programs can also dynamically change an RTID's attributes. For example, if a single phase in a program needs gang scheduling, the program can specify it for just that one phase.

```
struct McrtRtidS;
typedef struct McrtRtidS * McrtRtid;

typedef struct {
    unsigned doLoadBalancing;
    unsigned width;
    unsigned minGangCount;
} McrtRtidAttrs;

McrtRtid mcrtCreateRtid(McrtRtidAttrs *attrs);
void mcrtFreeRtid(McrtRtid rtid);
McrtThread *mcrtThreadCreate(McrtThreadFunc f,
                             void *args,
                             McrtRtid rtid);
void *mcrtAssociateRtidWithHwt(McrtRtid rtid,
                               int hwt);
void mcrtAssociateRtidWithData(McrtRtid rtid,
                               void *addr);
void *mcrtMallocForRtid(size_t size,
                        McrtRtid rtid);
```

Figure 1: The RTID placement interface

3.2 The RTID placement interface

This section describes the interface that our placement framework presents to applications. Although we present this framework as an extension to the threading and scheduling support of the McRT many-core runtime system [14], nearly the same interface could be added to other operating systems. For example, supporting these extensions on Linux would require minor modifications to the scheduler and the Native POSIX Threads Library (NPTL) [6].

Figure 1 describes our interface for specifying placement. The interface is relatively short and includes some RTID-related types, two RTID management functions, a thread creation function, and a few explicit data and thread placement functions.

- `McrtRtid` values, which are opaque, represent RTIDs.
- `McrtRtidAttrs` structures specify attributes for RTIDs. With a `McrtRtidAttrs` structure, a program can specify a number of items to either control or guide the scheduling of the RTID's threads. Most programs set the `doLoadBalancing` field to 0 to indicate that data proximity is more important than load balancing. However, this field can also be set 1 to have the scheduler emphasize load balancing. The `width` field gives the expected number of simultaneous threads sharing the RTID, or 0 if unknown. Finally, if `minGangCount` is greater than 1, the RTID's threads will be gang scheduled; otherwise its threads will be run near each other, but not necessarily at the same time.
- RTIDs are created using the `mcrtCreateRtid`

function. Note that its `attrs` argument is a pointer to an `McrtRtidAttrs` structure instead of the structure itself; this allows the program to update the RTID's attributes by simply modifying the structure in memory; the changes will take effect when the scheduler next runs.

- The `mcrtFreeRtid` function frees any resources associated with the RTID.
- The `mcrtThreadCreate` function creates a new thread. If the new thread should be associated with an RTID, that RTID is given by the `rtid` parameter; otherwise, 0 (i.e., `NULL`) should be given.
- Explicit placement support is provided by the `mcrtAssociateRtidWithHwt` function. This pins an RTID to a particular HWT so that its threads will always run on that hardware thread.
- Two functions control data placement. The first function, `mcrtAssociateRtidWithData`, indicates that the RTID's threads should be run on HWTs sharing fast caches that can hold data at or close to the given address. The second, `mcrtMallocForRtid`, is a variant of `malloc` that allocates the data "close" to the given RTID. It has one of the HWTs executing the RTID's threads allocate the data.

4 Discussion

This section summarizes how our framework addresses a number of issues that would be faced by any scheduler design. First, there is the question of what information the application programmer should provide to guide thread scheduling. Our approach is to require only high-level information unless explicit thread placement is being used. We expect the developer to identify the threads that share common data or locks and to assign those threads the same RTID. The scheduler will then assign HWTs to these threads in such a way as to reduce cache misses. To help guide the scheduler, we also allow the user to provide a small amount of additional information such as whether to emphasize load balancing or proximity to data, and whether gang scheduling is required. Because our scheduling framework does not require the developer to supply architecture-dependent information, such as the specific cache level threads should share, it helps to preserve the application portability that general-purpose developers need.

Another issue is whether to support explicit placement of threads and data. Although our scheduling framework is oriented towards automatic thread scheduling, we also support explicit placement by allowing the programmer to bind an RTID to a particular HWT. In this way, our API for explicit placement is consistent with that for automatic placement. However, the framework does not make any performance guarantees if programs mix explicit and automatic placement for their threads. In addition our framework also supports explicit data allocation by allowing the client to specify that data be placed close to an RTID. In this case the allocation is performed on an HWT that is executing threads associated with the RTID.

There is also the question of whether the framework can optimize thread scheduling on a range of different systems. In the past, parallel programs typically had to be customized for a particular system to get good scalability and performance on that system. This has been true, for example, for HPC programs. However, a program tuned to run well by itself on a particular system may not perform well if it is only one of several programs running simultaneously. Since general-purpose systems rarely run a single application at a time, we decided in favor of portability and ease of programming. Our approach is to have the developer guide placement using RTIDs and then have the scheduler use this information to place threads to achieve the best possible performance. While this approach may not be able to achieve the high performance that programs hand-tuned for a particular system might achieve, it should provide good performance across a range of programs on different architectures.

5 Scheduler design

Our framework uses an extension of the scheduler in McRT [14]. This is a highly configurable task-queue-based scheduler, and clients may specify the number of task-queues, the task-queue to HWT mapping, as well as the scheduling policies. Our initial implementation uses one task-queue per processor core and a combination of the scheduler's existing work-stealing and work-distribution scheduling policies. This configuration allows us to maximize sharing through the L2 (last-level) caches, while making use of the existing scheduling policies to achieve good load balancing. In general, our goal is to improve performance by minimizing contention for different resources and by maximizing HWT usage.

When new software threads are created, the scheduler tries to distribute them to task queues based on the load in the system. If the number of threads is less than the number of HWTs, the scheduler tries to improve throughput by channeling work to unused HWTs. Once every HWT is in use, the scheduler switches to *saturation mode* where scheduling is increasingly guided by RTIDs. When a thread is created in saturation mode, its RTID is compared to those of threads already in the system. If that RTID already exists, the scheduler will place this thread on a task queue for a core that shares as many levels of cache as possible with other threads sharing the

RTID. If that RTID is new, the thread is placed in the least loaded task-queue. The scheduler also uses work-stealing to try and keep the system's load balanced. Idle HWTs steal work from other tasks queues in the system. As a result, threads may migrate across HWTs. However, we expect thread migration to be relatively inexpensive on many-core processors.

As threads continue to be added to the system, eventually threads of multiple RTIDs will be scheduled on the same core. Our scheduler optimizes the order in which threads are run by grouping them according to RTID. When an RTID is scheduled, the threads within that group are run round-robin for a period of time. To ensure fairness, RTIDs are occasionally preempted and replaced at the end of the task queue. This approach is an extension to CPU affinity in which schedulers place threads on the same HWT they last ran on in order to reuse data in local caches. Although CPU affinity is likely to be less important on many-core systems, where the fast interconnect reduces the cost of an on-die memory miss, it is still likely to be important for many applications.

6 Related work

Many thread libraries on both Windows and POSIX operating systems give clients some control over thread scheduling. For example, POSIX threads allow clients to specify a scheduling policy and priority for their threads; standard scheduling policies include whether to use a first in-first out or round-robin scheduling policy. Furthermore, thread libraries typically provide some support for binding a thread to one or more processors.

Solaris provides a locality-oriented mechanism to optimize performance in NUMA systems. *lgroups* (locality groups) [5] represent a collection of CPUs and memory resources that are within some latency of each other. Lgroups are hierarchical and are created automatically by Solaris based on the system's configuration and different levels of locality. The system assigns each newly-created thread a home lgroup that is based on load averages, although applications can give a thread a different home lgroup. Lgroups help to control where threads and memory are allocated: when a thread is scheduled to run, it is assigned the available CPU nearest to the home lgroup, and memory is gotten either from the home lgroup or some parent lgroup. As a result, lgroups can be used to improve the locality of an application's threads, but are more restricted than RTIDs, which can also be used to specify gang scheduling and other scheduling-related information.

Since RTIDs are used to group related threads they may be compared with the *process groups* supported by both Windows and POSIX operating systems. Process groups allow a group of processes to be treated as a unit that can be identified by a *process group ID*. While process group IDs have some similarities to RTIDs, they are primarily used to control the distribution of signals and not specifically intended to improve performance.

Recent work on scheduling algorithms for many-core processors includes that of Fedorova [7]. Her cache-fair thread scheduling algorithm provides fairer thread schedules and greater performance stability on shared-cache multi-core processors by continually adjusting, for each thread, its time quantum to ensure that that thread has a fair miss rate in the L2 (i.e., last-level on-die) cache. The target-miss-rate algorithm helps to keep processor utilization high by achieving a target L2 cache miss rate. It does this by dynamically identifying threads that provoke the highest miss rates and lowering their priorities. Anderson, Calandrino, and Devi [2] apply a cache-aware thread co-scheduling algorithm to real-time systems. This algorithm reduces L2 contention by avoiding the simultaneous scheduling of problematic threads while still ensuring real-time constraints.

There is a large body of related work on thread scheduling to improve application performance on SMT processors. For example, Parekh, Eggers, Levy, and Lo [13] demonstrate significant performance speedups from scheduling algorithms that uses PMU feedback to select the best threads to schedule together. They found the best performance from an algorithm based on IPC feedback. Fedorova's non-work-conserving scheduler [8], which improves application performance by reducing contention for shared processor resources like the L2 (last-level on-die) cache. The algorithm uses an analytical model to dynamically determine when it makes sense to run fewer threads than the number of HWTs.

An especially relevant paper on SMT scheduling is that by Thekkath and Eggers [15], who found that placing threads sharing data on the same processor did not improve cache performance or execution time. However, their study did not consider more threads than HWTs and their threads were fairly coarse-grained. We would like to investigate whether their results still hold today for both many-core processors and emerging parallel applications in domains such as streaming, media processing and *RMS* (recognition, mining, and synthesis) applications [12].

7 Conclusion

Many-core systems will encourage the development of parallel, general-purpose applications. As the number of programmers for these applications increases, the need will grow for simpler but still effective ways of using large-scale parallelism. This paper describes a scheduling framework that tries to achieve good performance for the majority of applications on different many-core platforms without compromising ease of programming. We

believe having applications provide high-level guidance to a scheduler through RTIDs will let the scheduler do a better job for most programs than if those programs did explicit thread placement.

We are in the process of completing the initial implementation of this framework. Several mechanisms such as thread stealing and support for multiple scheduling policies are already a part of the McRT runtime, and our framework implementation is an extension of the current McRT scheduler. Our near-term plans are to evaluate the framework when used to run a number of our benchmarks, and then to experiment with different scheduling policies.

Future work includes the addition of several features our scheduling framework does not currently support. This includes support for multiple many-core processors, non-homogeneous processors, thread priorities, hierarchical RTIDs, multiple RTIDs for each thread, and NUMA distributed address spaces. In addition, we also plan to investigate integrating different feedback-based mechanisms into our scheduler to try to improve performance, system utilization, and fairness. In particular, we want to study how we could use PMU-based information about thread and system behavior to augment the RTID information from users.

8 Acknowledgments

We want to thank Jim Stichnoth, Neal Glew, and Youfeng Wu for their help in preparing this paper. We also thank the anonymous referees for their helpful feedback and suggestions.

References

[1] ALLEN, E., CHASE, D., HALLETT, J., LUCHANGCO, V., MAESSEN, J.-W., RYU, S., STEELE, G., AND TOBIN-HOCHSTADT, S. The Fortress language specification, version 1.0beta.

[2] ANDERSON, J. H., CALANDRINO, J. M., AND DEVI, U. C. Real-time scheduling on multicore platforms. In *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)* (Washington, DC, USA, 2006), pp. 179–190.

[3] CHAMBERLAIN, B., CALLAHAN, D., AND ZIMA, H. Parallel programmability and the Chapel language. *Int. Journal of High Performance Computing Applications* (Exp 2007).

[4] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05* (2005).

[5] CHEW, J. Memory placement optimization (MPO). http://opensolaris.org/os/community/% linebreak[0]performance/numa/mpo_update.pdf, Jul 2006.

[6] DREPPER, U., AND MOLNAR, I. The native POSIX thread library for Linux. http://people.redhat.com/drepper/nptl-design.pdf.

[7] FEDOROVA, A. *Operating System Scheduling for Chip Multithreaded Processors*. PhD thesis, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA, Nov 2006.

[8] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. A non-work-conserving operating system scheduler for SMT processors. In *Proc. USENIX 2005 Annual Technical Conference* (Anaheim, California, April 2005).

[9] INTEL. Teraflops research chip. http://www.intel.com/research/platform/terascale/teraflops.htm, Feb 2007.

[10] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro 25*, 2 (Mar 2005), 21–29.

[11] LEWIS, B., AND BERG, D. J. Multithreaded programming with Pthreads. *Prentice Hall* (1998).

[12] LIANG, B., AND DUBEY, P. Recognition, mining and synthesis. *Intel Technology Journal 9*, 2 (May 2005).

[13] PAREKH, S., EGGERS, S., AND LEVY, H. Thread-sensitive scheduling for SMT processors. *University of Washington Technical Report 2000-04-02* (2000).

[14] SAHA, B., ADL-TABATABAI, A., GHULOUM, A., RAJAGOPALAN, M., HUDSON, R., PETERSEN, L., MENON, V., MURPHY, B., SHPEISMAN, T., SPRANGLE, E., ROHILLAH, A., CARMEAN, D., AND FANG, J. Enabling scalability and performance in a large scale CMP environment. *EuroSys* (March 2007).

[15] THEKKATH, R., AND EGGERS, S. J. Impact of sharing-based thread placement on multithreaded architectures. In *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture* (Los Alamitos, CA, USA, 1994).

Automatic Mutual Exclusion

Michael Isard and Andrew Birrell
Microsoft Research, Silicon Valley

Abstract

We propose a new concurrent programming model, *Automatic Mutual Exclusion (AME)*. In contrast to lock-based programming, and to other programming models built over software transactional memory (STM), we arrange that *all* shared state is implicitly protected unless the programmer explicitly specifies otherwise. An AME program is composed from serializable atomic fragments. We include features allowing the programmer to delimit and manage the fragments to achieve appropriate program structure and performance. We explain how I/O activity and legacy code can be incorporated within an AME program. Finally, we outline ways in which future work might expand on these ideas. The resulting programming model makes it easier to write correct code than incorrect code. It favors correctness over performance for simple programs, while allowing advanced programmers the expressivity they need.

1 Introduction

Since the 1960's, system programmers have been trying to create and maintain *concurrent programs*. Initially, this desire arose as a means to allow the computer to continue useful program execution while some other part of the overall computation was unable to make progress because of a peripheral device. It also arose with the development of systems that shared the processor between multiple computations. Those who thought carefully about the problem abstracted it into the concept of multiple concurrent *threads* of execution (typically called *processes* at the time). In most analyses (CSP being an exception) the threads communicate through shared memory, and cooperate by achieving mutual exclusion on access to the memory [4]. Over time, this has become standardized as concurrent threads using shared memory and mutexes.

Unfortunately, in the real world most programmers find it difficult to use threads, shared memory, and mutexes correctly. In practice most mainstream applications tend to exhibit too little concurrency (they deadlock, or the user interface freezes during a network I/O, or they perform poorly on a multi-processor), or else they exhibit too much concurrency (by providing an incorrect answer, especially on a multi-processor, and usually non-deterministically).

A variety of problems with mutexes seem to cause much of this difficulty. One is the constraint that all the mutexes in a program have a partial order in which they are acquired, in order to prevent deadlocks. While this is a manageable constraint in systems built by small teams of programmers, it can be very difficult to meet in large systems, and extremely difficult over the long maintenance life of such systems. It is especially difficult to debug situations in which such deadlocks occur with very low probability, or where they are not formally a deadlock at all, just an unpleasantly long delay (such as waiting for a network service with a mutex held).

A second difficulty is often described as *lack of composability*. If a programmer wishes to layer an atomic operation on top of an existing abstraction (for example, adding an atomic "move" operation to move an item between two hash tables), this is difficult to do without access to the internal mutexes and mutual exclusion algorithms of the lower-level abstraction.

Finally, the existing mutex designs are fundamentally fragile: performance requirements (real or imaginary) create an incentive for the programmer to be "clever", and to minimize the use of mutexes, or their scope, or the amount of data protected by them. While this can be achieved correctly by a sufficiently clever programmer, it makes for tricky and un-maintainable programs — especially after the clever programmer has moved on to other projects.

Transactional memory (in software today, in hardware later) shows promise as an alternative to cooperation through mutexes [5]. It is generally presented to the programmer as *atomic blocks*, where the programmer delineates a region of the program to be executed as a transaction. The runtime system takes responsibility for executing the program in such a way as to have the same semantics as if the atomic blocks had been executed in some serialized order.

Serialized transactions have an appealing simplicity, and atomic blocks alleviate quite a lot of the problems of mutexes. The program no longer needs a partial order on its mutual exclusion calls, because if a conflict occurs the transaction machinery will detect it, and will almost certainly fix it by a suitable series of transaction aborts and retries. Atomic blocks also fix the composability problem: in the hash table "move" example, wrapping an atomic block around the get-insert-delete sequence will create the desired atomicity.

Unfortunately, just providing atomic blocks will not solve all our mutual exclusion problems. Fundamentally, they still require the programmer to decide what regions of the program need protection, and what data must be protected. Furthermore, the defaults are the

wrong way round: when a programmer cleverly decides that it is correct to narrow a mutual exclusion range, or to exclude some data from it, the programmer does so by removing text from the source code. We believe that this makes programs harder to understand, and quite difficult to maintain. For large systems (the only ones that are really difficult in this area), this is a critical defect. Put another way: with atomic blocks, the default and simplest program is one with no atomic blocks. Unfortunately, it is also the one least likely to work correctly.

The highest-level goal of this paper is to point out that the mechanism of transactional memory can be presented to the programmers with language constructs other than atomic blocks. We propose one such arrangement, but we have no doubt that others will follow.

Our proposal here is a new programming model, *Automatic Mutual Exclusion (AME)*, which we believe (hope) will cause programmers to be more likely to create applications that are concurrent, correct, and responsive, and which will remain so over the applications' life cycles. We do this by reducing the programmers' responsibility for concurrency and synchronization. By default, an AME program is correctly synchronized: if the programmer thinks not at all about mutual exclusion, there will be no data races. We then allow the programmer to take this correctly synchronized program and optimize it, by *adding* to the source code, not subtracting from it. Optimizing the concurrency behavior of an AME program requires actions where the programmer explicitly declares the places where the optimizations occur, in a way that we believe will be maintainable.

2 Asynchronous method calls

We begin by describing a simplified AME programming model that supports basic concurrent event-based programming. The remainder of the AME model is described in sections 3 and 4.

In this simplified model, running an AME program consists of executing a set of *asynchronous method calls*. The AME system guarantees that the program execution is equivalent to executing each of these calls in some serialized order (i.e., atomically). AME achieves concurrency by overlapping the execution of the calls, subject to maintaining this guarantee. The program terminates when all its asynchronous method calls have completed.

Initially, the set consists of a call of `main(…)` initiated by the AME system. Within an asynchronous method call, the program can create another asynchronous method call by executing:

```
async MethodName(MethodArguments);
```

In terms of the formal semantics of the program, the newly created asynchronous method call will be serialized after the current one. In terms of program structure, the `async` construct is reminiscent of forking a thread in thread-like systems, or posting an event in event-based systems (but subject to the serialization guarantee).

While the semantics are serialized execution, naturally the AME system will attempt to execute the set of available asynchronous method calls concurrently, within the available resources and subject to strategies (TBD!) that prevent excessive transaction aborts.

To achieve our serialization guarantee, our basic implementation is that each asynchronous method call will be executed by the AME system as an STM (or HTM) transaction, within a thread from a pool provided by the AME system. However, the semantics say nothing about transactions: if the AME system can determine that a cheaper synchronization scheme (such as mutexes, or no locking at all) will achieve the serialization guarantee for a particular program execution, it is free to use that scheme.

When a transaction initiates other asynchronous method calls, their execution is deferred until the initiating transaction commits. If the initiating transaction aborts, they are discarded. When it commits, they are made available for execution (in an indeterminate order).

Within an asynchronous method call, the program is not permitted to take actions with side-effects that the AME system cannot undo: this enables an implementation using transactional memory. In particular, I/O activity cannot occur (but we'll deal with that below).

This much of the design allows programming with concurrent non-blocking asynchronous method calls (or events). The program is correct, in that the calls can share memory without any risk of races. The calls can execute concurrently when possible, and the AME system will ensure that the result is a valid serialization of the events. The programmer has written no synchronization code.

We still need give the programmer optimization mechanisms: some control over transaction scheduling (section 2.1), and some way to split up transactions to reduce the frequency of aborts or to enable rational program structure (section 3). We also need to provide for legacy code, and for code with non-abortable side-effects such as actual I/O (section 4).

2.1 Blocking an asynchronous method

An asynchronous method may contain any number of calls to the system-supplied method:

```
BlockUntil(<predicate>);
```

From the programmer's perspective, the code of an asynchronous method executes to completion only if all the executed calls of `BlockUntil` within the method have predicates that evaluate to `true`.

`BlockUntil's` implementation does nothing if the predicate is `true`, but otherwise it aborts the current transaction and re-executes it later (at a time when it is likely to succeed). This is like `Retry` in some systems.

For example, a blocking read from a shared queue could be implemented as:

```
BlockUntil(queue.Length() > 0);
data = queue.PopFront();
```

Notice that the AME system has a lot of information available when `BlockUntil` is called with `false`: it can in principle determine what non-local memory affected the evaluation of the predicate, and it can determine when other asynchronous method calls later modify that memory. We envisage taking advantage of this to optimize the scheduling of the transaction retry.

2.2 Examples and discussion

At this point we introduce some example fragments that illustrate common concurrent idioms. By convention variables that live in the shared heap begin '`g_`' (for "global"). The first example performs overlapped reading from a file, where 4 blocks are in flight at any given time (error handling is ignored for simplicity, and we elide the details of how I/O is performed within the file library, since that needs section 4):

```
void OpenRead(FileName name) {
  File f = StartOpen(name);
  async StartRead(f);
}

void StartRead(File f) {
  BlockUntil(f.Opened);
  g_nextOffset = 0;
  g_nextOffsetToEnqueue = 0;
  for (int i=0; i<4; ++i) {
    ReadBlock block = new ReadBlock;
    block.offset = g_nextOffset;
    block.file = f;
    g_nextOffset += block.size;
    f.StartRead(block);
    async WaitForBlock(block);
  }
}

void WaitForBlock(ReadBlock block) {
  BlockUntil(block.ready &&
             g_nextOffsetToEnqueue ==
               block.offset);
  if (block.EOF) {
    g_endOfFile = true;
  } else {
    g_queuedBlocks.PushBack(block);
    block.offset = g_nextOffset;
    g_nextOffset += block.size;
    block.file.StartRead(block);
    async WaitForBlock(block);
  }
  g_nextOffsetToEnqueue += block.size;
}
```

The second example simulates a fragment of a computer game, implementing the logical thread of control for a particular character that is moving autonomously and interacting with its environment:

```
void StartZombie() {
    Zombie z;
    z.Initialize();
    /* schedule the first move */
    async UpdateZombie(z);
}

void UpdateZombie(Zombie z) {
  Time now = GetTimeNow();
  BlockUntil(now - z.lastUpdate >
             z.updateInterval);
  z.lastUpdate = now;
  MoveAround(z);
  if (Distance(z, g_player) <
    DeathRadius) {
    KillPlayer();
  } else {
    /* schedule the next move */
    async UpdateZombie(z);
  }
}
```

The final example illustrates a data-parallel computation that processes every item notionally "in parallel" and inserts the results into an output list whose ordering is undefined:

```
void DoBatch(Queue inQ, Queue outQ) {
  BlockUntil(inQ.Length() > 0 ||
             g_finished);
  while (inQ.Length() > 0) {
    Item i = inQ.PopFront();
    async DoItem(i, outQ);
  }
  if (!g_finished) {
    async DoBatch(inQ, outQ);
  }
}

void DoItem(Item i, Queue outQ) {
  DoSlowProcessing(i);
  /* Writing to outQ here would serialize the slow
     processing, because of contention on outQ */
  async DoOutput(i, outQ);
}

void DoOutput(Item i, Queue outQ) {
  outQ.PushBack(i);
}
```

Note that none of these examples require the programmer to make a determination of what shared state must be protected; the AME system protects the state automatically. The programmer is not tempted to consider, for example, leaving the code in `StartRead` after the `BlockUntil` call unprotected. We believe this automatic concurrency protection will be extremely valuable in complex and long-lived programming projects.

The event-based AME programming model is attractive because it makes it extremely difficult to write a "fine-grain" concurrency bug: every method call is always executed in its entirety or not at all. We also believe that the mental model of serialized execution is among the simplest concurrency abstractions for programmers to understand. Higher level races are still possible (e.g., by assuming that some state is preserved unmodified between asynchronous method calls).

3 Fragmenting an asynchronous method

With the facilities of the previous section, the programmer will inevitable be faced with a situation where a conceptually single asynchronous method call must be split up. In the simplest cases, this arises when the call creates too many memory conflicts with other calls, causing too many transaction aborts. (This is analogous to holding a mutex for too wide a range of the program.)

As explained by several previous authors [1,3], other cases arise that require splitting up events in a pure event-based model, producing program structure that can be unpleasant, and unstable. For example, if a previously non-blocking method call is modified to require a blocking action (e.g., a hash table is modified to use disk storage instead of main memory), the event-based style would require that the method, and all of its callers, gets split into two separate methods (a request and a response handler). This is sometimes referred to as "stack ripping".

Our solution to both of these problems is to allow an asynchronous method call to contain one or more calls to the system method `Yield`. A `Yield` call breaks a method into multiple *atomic fragments*. This is similar to breaking an atomic block into multiple adjacent blocks, except that our atomic fragments are delimited dynamically by the calls of `Yield`, not statically scoped like explicit atomic blocks.

With this enhancement, the overall execution of a program is guaranteed to be a serialization of its atomic fragments. We (intend to) implement `Yield` by committing the current transaction and starting a new one.

A `BlockUntil` call only blocks execution of the current atomic fragment (the code following the most recent `Yield()`), or equivalently, it only retries the transaction begun after the most recent `Yield`.

A `Yield` call can occur within any method, including libraries. Since `Yield` splits atomic executions, it is critical that a caller be aware of this possibility: the caller's own shared state becomes visible to other asynchronous method calls, and might be changed by other asynchronous method calls. We require that any method containing a `Yield` makes this explicit statically, by having a type signature such as the following:

```
ReturnType MethodName(Args) yields {…}
```

Any synchronous call to such a method must be decorated:

```
int foo = Method(x) yielding;
```

Of course a function that calls a `yielding` method must itself be marked `yields`.

Notice the effect on program maintenance. If a previously non-yielding library changes its implementation to use `Yield`, then the library's callers will get a compilation error because of the lack of a `yielding` annotation. The callers must then either determine that it is correct to expose their shared state (i.e., their invariants are true), or they must remove the offending call.

Here as elsewhere in the design we have chosen to require that the programmer makes explicit the places where races might occur, leaving the default case

correctly synchronized, despite the inconvenience this causes.

We might be concerned that most library calls will contain *some* code path that leads to a `yielding` method. The result would be that most code would become tainted as `yields`, eliminating any useful information content from the annotation. However, there is no incentive to insert a call to `Yield` in a library method unless that method is either extremely long-running, or must block, e.g. waiting for synchronous I/O. Consequently we view the potential for a library to become tainted as a strength rather than a weakness. Some of the worst performance bugs arise from calling a method that is normally fast, but that blocks occasionally due to rare corner cases. Under our model, the programmer will know to avoid such calls or else plan for the possibility of a slow execution.

At this point we can rewrite our zombie and queue examples more concisely:

```
void RunZombie() yields {
  Zombie z;
  z.Initialize();
  do {
    Yield();
    Time now = GetTimeNow();
    BlockUntil(now - z.lastUpdate >
               z.updateInterval);
    z.lastUpdate = now;
    MoveAround(z);
    if (Distance(z, g_player) <
        DeathRadius) {
      KillPlayer();
    }
  } while (Distance(z, g_player) >=
           DeathRadius);
}

void DoQueue(Queue inQ,
       Queue outQ) yields {
  do {
    Yield();
    BlockUntil(inQ.Length() > 0 ||
               g_finished);
    while (inQ.Length() > 0) {
      Item i = inQ.PopFront();
      async DoItem(i, outQ);
    }
  } while (!g_finished);
}

void DoItem(Item i,
       Queue outQ) yields {
  DoSlowProcessing(i);
  Yield();
  outQ.PushBack(i);
}
```

In addition to the obvious structural improvements, notice that the zombie `z` variable is now allocated on the thread's private stack. In a transactional environment thread-private variables are more efficient, and making this clear may make optimization easier.

4 Unsynchronized Fragments

There are times when automatic mutual exclusion prevents the program from doing what it needs to do. The two obvious cases are access to legacy code that doesn't use this machinery, and code with non-abortable side-effects such as I/O. To support these we allow the following:

```
unprotected { … }
```

This construct terminates the current atomic fragment (typically by committing the current transaction), then executes the inner block, then starts a new atomic fragment. Any method that uses `unprotected` must be flagged as `yields`.

The typical pattern for a region of the program that wants to perform I/O (probably by calling legacy libraries, but perhaps by calling the kernel directly, or even by writing to device registers) will be as follows. Within an atomic fragment, the program decides on the details of the I/O operations and stores them on a thread-local queue. It then uses `unprotected` to execute code that extracts the requests from the thread-local queue and passes them into the actual I/O system. Typically, this dance will be performed within some AME library code or wrappers. The library calls that perform synchronous I/O will consequently be marked as `yields`, as expected.

However, an asynchronous I/O library (such as the `StartOpen` and `StartRead` methods used in the example in Section 2.2) does not need to have its methods marked as `yields`, because the library internally defers the actual I/O calls by invoking them through asynchronous method calls.

Notice that since this particular pattern of using `unprotected` touches only thread-local and non-transacted memory, it is not subject to the issues of privatization common in transactional memory designs when interacting with non-transactional code. Other uses of `unprotected` might have privatization problems, and we are considering making them illegal.

5 Discussion

This proposal is not about inventing fundamentally new semantics for concurrency. Primarily, we are exploring a new way to present our existing mechanisms to the programmer. We intend that the programmer can write straightforward code first, and achieve a good level of concurrency with little risk of races. As the program develops, it can be optimized to achieve better concurrency, using mechanisms that flag the places where concurrency has been improved at the risk of introducing races. We encourage correctness first, performance second, and maintainability always. We support a non-blocking event-driven style, or a blocking procedural style with `Yield`, or any convenient combination.

The design is intended for real programs (though limited by today's STM performance). We intend to handle programs that perform disk and network I/O and that interact with the user. We intend to be able to handle large systems, with long lifetimes.

One way to view this design is to compare it with either single-threaded cooperative multi-tasking systems, or with single-threaded event based systems such as the JavaScript side of AJAX. Those programming models are similar to AME, except that AME can execute the program with real concurrency, utilizing real multi-processors — with very little extra effort from the programmer.

There are numerous research questions that we see arising from the AME proposal. Most obviously, we need to implement AME, develop some real experience, and determine whether it is in fact useful. We have a good STM implementation available to us, and we plan to modify our compiler to support the AME extensions instead of explicit atomic blocks.

Critical to AME performance is the scheduling of transactions so as to minimize the amount of work that will be aborted. We are hopeful that `BlockUntil` will help in this (more so than a simple `Retry` statement).

There are optimizations available in some cases of `BlockUntil`, especially where it occurs as a guard at the start of an atomic fragment (for example, we might not need to use transactions for those cases). We are considering whether to restrict its use to just those situations.

Transactional memory designs will continue to be impractical until the system can optimize by eliminating transactional overhead for memory accesses that are in fact thread-private. We have some tentative but incomplete ideas for doing this.

Our `Yield` operation could be enhanced by having the programmer specify a subset of the transactional variables that should be modifiable during the yield. This would enhance correctness by allowing the system to report an error if some other transaction attempts to modify the non-modifiable state.

Overall, we are excited by the possibility of the AME ideas. We believe they make it much more likely that programmers will create correct, efficient, and maintainable concurrent programs.

6 References

There are many important works in this area. We have assembled the 20 that we found most useful on a web page [4]. Below, we cite only the ones most specific to this paper.

1. Adya, A. et al "Cooperative Task Management without Manual Stack Management", Proc. Usenix 2002 Annual Technical Conference, June 2002.
2. Bacon, D. et al "The 'Double-Checked Locking is Broken' Declaration", http://tinyurl.com/1rja, viewed December 2006.
3. Von Behren, R et al "Why Events Are A Bad Idea", Proc. 9th Workshop on Hot Topics in Operating Systems, May 2003.
4. Birrell, A. "A Selected Bibliography of Concurrency", http://birrell.org/andrew/concurrency/, December 2006.
5. Harris, T. "Composable Memory Transactions", Proc. 10th Symposium on Principles and Practice of Parallel Programming, June 2005.

Hype and Virtue

Timothy Roscoe* Kevin Elphinstone† Gernot Heiser†‡
National ICT Australia

Abstract

In this paper, we question whether hypervisors are really acting as a disruptive force in OS research, instead arguing that they have so far changed very little at a technical level. Essentially, we have retained the conventional Unix-like OS interface and added a new ABI based on PC hardware which is highly unsuitable for most purposes. Despite commercial excitement, focus on hypervisor design may be leading OS research astray. However, adopting a different approach to virtualization and recognizing its value to academic research holds the prospect of opening up kernel research to new directions.

1 Introduction

Are hypervisors really a disruptive technology? Both the IT industry and the academic OS research community have devoted much attention recently to virtualization, in particular the development of hypervisors for commodity hardware, and commodity hardware support for them.

Virtualization has been touted in popular articles as a disruptive technology, and indeed as "the new foundation for system software" [6]. A recent spirited debate has centered on the claim that the hypervisor-based approach to system software fixes most of the perceived flaws of microkernels while retaining their apparent advantages [13, 14].

While the importance of the current wave of virtualization technology seems clear from a commercial standpoint, in this paper we critically examine whether hypervisors represent an equally disruptive factor for the OS research community. One might ask, to coin a phrase, "are virtual machine monitors OS research done right?"

We argue that this is not the case at present, and that most current research based around virtualization is not very different (if at all) from the kinds of problems the community has always worked on. However, we *do* feel that virtualization presents truly interesting directions for academic research (as opposed to product development or business models), both as an enabler for new ideas, and as a source of a new class of problem. We lay out some of these directions towards the end of this paper.

A challenge with any technological development which creates intense interest simultaneously in both academia and venture capital circles is separating the long-term scientific and engineering questions traditionally relegated to academic research from the short-term issues closely tied to particular business models, contexts, and in some cases individual companies.

This is an unashamedly academic paper, deliberately bracketing short-term commercial pressures to concentrate instead on longer-term research questions in OS design. We do not wish to devalue short-term research strongly embedded in current products and markets, but we emphasize that it is not our concern here.

In the next section we compare hypervisors to other kernels from a long-term research perspective (rather than focusing on their short-term applications) and in Section 3 critique the new system interface offered by hypervisors. In Section 4 we identify an approach to building and using virtualization technology to move academic OS research along by freeing it from some of the business-oriented constraints that have dogged the field for some time. In Section 5 we outline a few possibilities that this view of virtualization opens up, and conclude in Section 6.

2 Much Ado

Current VM-related research falls into two areas: building better hypervisors, and novel applications for them.

Our target in this paper is the former, though first we remark in passing that a number of novel applications for hypervisors are either (admittedly useful) tools for writing existing operating systems, or ingenious workarounds for the deficiencies of the guest OS – the ideas are important, but a well-designed OS interface would make their implementation much easier, and they don't investigate what a radically new operating system design might achieve.

What the VMM is providing here is a means to get the work done without changing an existing guest OS, perhaps because such a job is beyond the capacity of a single PhD student or does not fit into a time frame dictated by upcoming publishing deadlines. These are important practical considerations, but we should also explore the

*Now at ETH Zürich, Switzerland
†Also at the University of New South Wales, Sydney, Australia
‡Also with Open Kernel Labs

long-term question of whether the combination of modified VMM and legacy operating system is a better solution than simply building a better OS in the first place.

Rather than treating these application ideas as simply neat VMM tricks, we should take them as new *requirements* for OS design and implementation. Resorting to a hypervisor to implement replay debugging or sophisticated security mechanisms, for instance, is a tacit admission that the current guest OS of choice cannot be practically evolved to support this functionality.

In the rest of this section, we critically examine the canonical tasks of a kernel – resource sharing, protection, abstraction of hardware, and communication – and try to establish what is genuinely different in a hypervisor versus a conventional kernel.

Sharing and Protection

Sharing is about controlling how multiple clients, in this case virtual machines, use the resources of the hardware. Protection is about ensuring that these clients do not unduly interfere with each other – by accessing each others' data, or affecting each others' performance by acquiring resources that system policy has not allocated to them.

We should ask whether approaches to sharing and protection in the new generation of hypervisors are in any sense novel. It is true that CPU resource allocation has been given a new context by the possibility of selling resources (packaged as VMs) in the form of "utility computing". However, almost all the solutions to this problem (mostly in the form of hypervisor scheduling algorithms) are quite old, lifted from the now-moribund field of multimedia systems (e.g. [8,17,23]). Data protection is achieved via per-VM MMU state – hardware access aside, essentially the same as address-space protection in a conventional OS, even if it is abstracted differently.

Communication

Communication between VMM clients is addressed very differently. Rather than borrowing solutions from monolithic and micro-kernels, hypervisor designs appear to define the problem away.

The argument [13] goes as follows: since the communication principals are complete operating systems in themselves, they are largely self-contained, much like library OSes over exokernels like Aegis [11] and Nemesis [17]. Consequently the VMM has little need to support the equivalent of fast IPC in microkernels, since the system as a whole has little need for shared servers. Furthermore, the kernels inside VMs are expecting to communicate via their own networking stack (since they assume they have a machine to themselves), and hence an emulated Ethernet should suffice for the small amount of local inter-VM communication they do need.

However, it seems that fast IPC performance is indeed important for hypervisors after all. A trend in hypervisor design is moving drivers into guest OS domains used as "driver servers", mirroring microkernels.

For example, the early design of Xen [1] resembled an exokernel, with low-level multiplexing of resources between relatively self-contained domains. Recently, Xen has morphed into an architecture where drivers run in separate domains [12] which function like driver servers in microkernels [18]. There are compelling reasons to do this, though they're not particularly technical. In particular, Xen no longer needs device drivers since it can use those written for another kernel by running them in copies of that OS.

Hypervisors hence now perform a lot of IPC in the normal execution of guest OSes. Fortunately, a wealth of research in this area seems directly applicable, both for synchronous IPC (useful for short, bounded-execution-time calls such as driver invocations), e.g. [19], and asynchronous messages (suitable for data transport), e.g. [2].

Abstraction

The key area where hypervisors differ from traditional kernels is abstraction: how resources are presented to the client. Rather than a view based on processes, threads, and address spaces, hypervisors aim at an abstraction which resembles the hardware enough to run guest OS kernels written for the physical machine.

Compatibility aside, this abstraction has a clear advantage: it can be made to correspond to a user-level application. Ironically, both traditional microkernels and monolithic systems lack an explicit kernel representation of a complete application. For example, Unix applications often span multiple processes (or even process groups), while multiple applications can also share use of a single process (as with the X server).

A VM, on the other hand, can both contain and isolate an application, by bundling the guest OS with the application – as one industry figure has put it, "operating systems are the new middleware" [5]. We believe this is highly significant, even if it has occurred in hypervisors almost by accident. Software is now being sold on this basis in the form of "virtual appliances".

Aside from providing backward compatibility with existing operating systems, which from this paper's perspective is commercially important but of little research interest, this new interface is the salient feature of hypervisors: it offers an explicit abstraction of an application, to which one can apply security policies, resource allocation, etc.

However, as researchers thinking critically we must ask: while this new interface might be a better abstraction, is it the best or most appropriate one?

3 Virtual Hardware as an API

Having looked at what might be different about hypervisor design from an OS research standpoint, we now look at the design of the interface that hypervisors provide to their clients: VMs, and ultimately applications.

The interface provided by hypervisors was not designed with applications in mind at all. It is based on hardware, with modifications (paravirtualization) to address the worst performance problems. Its use for virtual appliances came after hypervisors had been around for some time, motivated by quite different reasoning (such as making parallel programming tractable [4]).

In fact, we argue that a paravirtualizing hypervisor is a bad choice for an OS interface, for a number of reasons.

Implementation complexity

A feature of the VMM-based approach is *less coding*, since to support an existing application, one simply runs the OS it expects underneath.

However, this simplicity comes at the cost of *more code*. There's now a lot of machinery on the path between the application and the (real) hardware, not directly contributing to the application's functionality (or duplicated in the stack).

The resultant bloat comes with its own security problems, since the system's trusted computing base increases in size, and its correctness becomes even harder to ascertain. Whereas previously an administrator had to be concerned with vulnerabilities in the underling OS compromising the application, now he or she needs to be worried about the application's guest OS, the VMM itself, device drivers in driver domains, and the guest OSes supporting these drivers: all of these components must now be kept up-to-date, and vulnerabilities in any of these components can jeopardize the application.

Interface complexity

It is sometimes claimed that the VMM approach leads to better system design because the VM interface is simple and low-level, leading to a more policy-free and verifiable system as a whole. This argument sounds appealing – after all, the systems research community we have always valued "elegance" and "simplicity" in our designs. Unfortunately, on close inspection there is absolutely no evidence for this.

Part of the trouble is that we can't say what we mean by "simple" or "elegant" designs. There are no agreed-upon metrics or definitions. This problem has been eloquently pointed out recently in networking [22].

A good case can be made that the low-level interface provided by hypervisors is more complex and harder to specify than typical OS ABIs (or abstract virtual machines like Java's VM or Microsoft CIL), and is hardly a move closer to formally-specified interfaces. PC-based virtual machines vary widely in supported instruction set extensions, available (virtual) hardware, physical memory layout, MMU functionality, interrupt delivery semantics, etc.

Furthermore, we know of no attempts to even informally specify this interface. At best, paravirtualization interfaces such as [24] attempt to capture the *differences* between the VM's ABI and that of the (unspecified) real hardware.

In contrast, interfaces to operating systems (both microkernels and monolithic systems) and language-based virtual machines have a long history of careful documentation [16, 20], standardization [9, 15], and more recently, formal specification [10].

An anonymous reviewer of an earlier version of this paper suggested there might be something "almost magical" about the PC hardware ABI responsible for its recent success – an intriguing notion worthy of further study.

We conjecture that there are some features of the PC ABI (for example, an upcall-based interface in the form of interrupts) which are well-suited to supporting complex applications, while other aspects (for example, the ia32 MMU design) present serious obstacles to development. Evidence for this comes in the form of which parts of the interface have been redesigned by implementers of paravirtualizing VMMs. The challenge is to take what we can from this ABI, but design a better one.

Performance and scalability

Full virtualization, considered purely as an OS ABI, results in remarkable performance and scalability penalties compared to more conventional kernel interfaces. Part of this is due to the semantic bottleneck of the hardware-like interface – it's hard for a guest OS to express complex requests efficiently across this interface, but part of the problem is the overhead of running a complete copy of an OS in each VM.

For example, Xen scales remarkably well considering its aims, but in no way compares to vanilla Linux in terms of the number of application processes runnable at a time, due almost entirely to memory overhead. Even when modified to support copy-on-write images of mostly-identical VMs as in the Potemkin project [25], they still come out as much more heavyweight than processes.

Paravirtualization addresses these problems, but only slightly: paravirtualization starts from "pure" virtualization and backs off the minimum necessary for roughly correct execution and adequate performance. This works, but only up to a point, since the structure of the OS inside the VM still constrains the paravirtualized interface: the extra functionality typically appears as device drivers, for example.

In defence of hypervisors, we might say that the lack of scalability is a deliberate and considered consequence of providing performance isolation (through memory partitioning, etc.) between virtual machines. This argument, of course, does not address our point here, namely that a VM is a poor interface to an OS kernel. Moreover, it also fails to acknowledge two additional points that we feel are critical: firstly, we have no good metrics for measuring isolation, and secondly, it is not clear how effective VMMs can be at performance isolation in the presence of cache contention, even on a uniprocessor. We return to these below.

Discussion

It is ironic (and somewhat tragic) that after years of OS research we should look to the designers of PC hardware for guidance in formulating a kernel ABI. The new hypervisors have added to the traditional OS interface (POSIX or Win32) a second: paravirtualized PC hardware at the bottom. Neither is really new, nor are the designs of these interfaces terribly interesting from a future research perspective.

4 The opportunity

In OS research, we should use virtualization for what it's good for. By virtualizing a commodity OS over a low-level kernel, we gain support for legacy applications, and devices we don't want to write drivers for. Other than that, we should avoid these interfaces and move on. Virtualization presents us with an opportunity to return to basic research in system organization.

We should not conflate the facility of virtualizing hardware with the decision to multiplex the hardware at this level, even though current hypervisors do this. Instead, we could design a simple, clear interface to a low-level resource multiplexer in addition to the virtualization interface in current monolithic hypervisors like Xen. This would provide the benefits of a well-specified API to a low-level kernel, combined with the optional ability to run legacy OSes on top, rather than mandating the hardware-emulation interface of current hypervisors.

We must be clear that we are proposing is to write new operating systems which can virtualize existing ones as a means to gain compatibility with legacy applications and to reuse drivers for some hardware. We are *not* proposing new operating systems targetted at running inside virtual machine monitors (though we agree VMMs can be useful as hardware emulators in the early stages of kernel development). The latter is of dubious value in understanding how to effectively multiplex machine resources, the basic reason for an OS in the first place. The former allows us to concentrate on this fundamental research problem without the distractions of compatibility.

This could be done in two ways. A monolithic hypervisor like Xen could be extended so as to provide a more direct interface to the Xen kernel's facilities via hypercalls. Alternatively, a modular approach would split off hypervisor functionality into an optional hardware virtualization layer running as a library over a small resource multiplexer with a clean, well-specified interface. This latter component might be described without irony as a "microkernel".

Punting support for legacy applications and device drivers to a virtualized guest OS might raise performance concerns, but from a research perspect these issues are illusory. A research operating system project can demonstrate good native device performance by simply not virtualizing drivers for the devices that the researchers care about, and writing drivers for enough devices to demonstrate to the community that the design is sound. An analogous argument applies to applications: all legacy applications can be run with acceptable performance (if not, this calls into question much of the value of virtualization), but more importantly we can allow very different applications to emerge, and port applications to the new design where maximum performance is important.

All this allows the research community to finally escape the straitjacket of POSIX or Windows compatibility, both in drivers and applications. Rob Pike [21] has estimated that about 90-95% of Plan9's development effort was occupied with compatibility (excluding drivers). We have the opportunity to concentrate on OS design without further concern for backward compatibility. The result might be research kernels which are actually usable for real work, a rare sight these days.

5 Disruptive virtualization

Done right, virtualization removes the problems of driver support and legacy application compatibility, leaving open the questions of what the low-level kernel looks like and what its API is. Research kernels can experiment and *obtain practical experience* with a number of different approaches in this space.

Many systems research directions, and hypervisors are no exception, are characterized by what one assumes to be fixed (for example, the hardware and the processor architecture) and what can conversely be changed. Virtualization gives us the opportunity to redefine OS interfaces at any level above the physical hardware, allowing many areas of OS design to be rethought. Here we make a present a few of the possible avenues to investigate.

Kernel and API design

Many kernel interfaces are designed for single-threaded C programs using explicit memory management. These interfaces are in many respects a poor match for modern high-level programming languages.

For example, a well-known problem is how to integrate garbage collection with virtual memory: a copying collector can page in large amounts of memory from disk only to deallocate it. Exposing control of hardware page tables to the garbage collector can greatly improve matters, but is hard to achieve in a Unix-like system. The design of a suitable, shared interface to the MMU is a challenge.

There is also an opportunity to rethink API design (particularly with regard to I/O and concurrency) in the light of transactional memory, concurrent hardware, and high-level language constructs such as parallel combinators.

It is hard or impossible to attack these problems effectively either above a virtual hardware layer, or inside a hypervisor that supports such a layer, since the virtual hardware interface gets in the way. A more appropriate use of virtualization is enable a radical redesign of the lowest layer in the system while preserving compatibility for legacy devices and applications.

A final open question is whether a modular microkernel-based virtualization design can be as efficient at virtualization as a monolithic hypervisor. Note that situation is different from traditional microkernel issues, since it's more about vertical communication between the guest OS and microkernel, rather than horizontal IPC between processes (which is much the same in both cases).

Implementation techniques

A related problem is building an assured kernel, that is, one where we are reasonably certain the interface and implementation conform to a set of well-specified behaviors. It is famously hard to formally specify the behavior of existing OS APIs, let alone verify that a C or C++-based implementation results in the specified behavior.

However, it is hard to either gain traction with a new language in the context of an existing kernel written in C, or conversely validate a language by building a new kernel with little chance of deployment. The approach in Section 4 allows kernel design and language research to proceed together and result in a deployable OS.

A number of groups are working on better language support for systems programming. For example, Ivy [3] aims at evolving C with safe and checkable extensions.

The `sel4` project is investigating an alternative approach: build a prototype model of a kernel in Haskell [7,10]. The model can be combined with a machine simulator to execute real application binaries under simulation to gain experience with the API design, while the use of a high-level language facilitates clean implementation and rapid design iteration. A machine-checkable formal specification in higher-order logic can also be extracted from the model and used to verify API properties. The final design can then be executed on real hardware via a port of the Haskell runtime or translated semi-automatically into a low-level systems programming language for a high-performance implementation.

Applications

A principal justification to perform research into different designs of OS (rather than improvements to Unix or Windows) is to enable new user-visible functionality. A large portion of this is new applications. Given the opportunity to radically rethink kernel and API design, it is natural to ask what applications might be enabled by such research.

This is a difficult question, since "killer apps" tend to emerge unexpectedly from new enabling technologies, and the current state of the art often frames thinking about new applications. There are classes of applications (for example, rich QoS-based multimedia applications) which motivated considerable OS-related research in the past, but which arguably failed because of the difficulty of integrating the techniques with existing mainstream systems.

At the very least, virtualizing a legacy guest OS and redefining the underlying kernel interfaces does strictly increase the space of feasible applications.

The need for metrics

As mentioned above, there are no good OS benchmarks for isolation or VMM scalability. In other areas of computer systems research (such as databases and filing systems) benchmarks have demonstrated benefit in evaluating solutions and comparing approaches.

Benchmarks are problematic in kernel design, in part because it is hard to devise metrics which are not tied to a particular interface, and an important aspect of what we do as OS researchers is devise better interfaces. But without objective measures of how well isolation kernels (VMMs, monolithic kernels, or raw microkernels) do their job, we cannot make meaningful comparisons.

More significantly, in the absence of well-designed benchmarks it will be hard to gauge whether performance isolation is feasible at all without hardware support. Anecdotal evidence suggests that contention for cache lines between processes can have a serious effect on application progress. As multiprocessors become the norm, this effect will become more pronounced and contention for main memory will become important as well. Any argument that VMMs (or any other isolation technology) is an improvement on the previous state of the art must be set in this context.

Another benefit of benchmarks and metrics is less quantitative: it prompts discussion of which performance dimensions are actually important. A wider question is, what kinds of metrics are suitable for research kernels, whether monolithic hypervisors or microkernel-based?

In the absence of metrics, OS research often appeals to notions of simplicity. But as we have seen, these are highly ambiguous: from one perspective, a paravirtualized hardware interface represents a new "narrow waist" in system software, but from another is maddeningly complex, ill-specified, and poorly designed for programmers.

6 Conclusion

This paper has argued that OS researchers these days face a choice. On the one hand, we can investigate better ways to implement existing interfaces, i.e. further tweaks to Unix, better hypervisor implementation, and better paravirtualization techniques in the guest OS. This is short-term, immediately applicable, commercially relevant, but cannot be described as disruptive, since it leaves most things completely unchanged.

On the other hand, we can recognize that virtualization techniques do not necessarily mandate a hardware-like VM interface or a (para)virtualized conventional OS above, even if this is all that current hypervisors support. As researchers thinking long-term, virtualization techniques might give us the freedom to look at alternatives to these two interfaces without having to give up existing application and hardware support.

This opens up a variety of avenues in OS research, including novel engineering methodologies, application for formal methods, better support for modern programming languages and processor architectures, and discussion of appropriate metrics for evaluating future systems. We have tried to suggest a few of these in this paper.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th SOSP*, October 2003.

[2] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol implementation in a vertically structured operating system. In *IEEE LCN'97*, pages 179–188. IEEE, November 1997.

[3] E. Brewer, J. Condit, B. McCloskey, and F. Zhou. Thirty years is long enough: Getting beyond C. In *Proc. HotOS-X*, June 2005.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proc. 16th SOSP*, 1997.

[5] S. Crosby. Personal communication, August 2006.

[6] S. Crosby and D. Brown. The virtualization reality. *ACM Queue*, 4(10), December 2006.

[7] P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proc. ACM SIGPLAN Workshop on Haskell*, September 2006.

[8] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. 17th SOSP*, 1999.

[9] ECMA International. *Common Language Infrastructure (CLI)*, 4th edition, June 2006. ECMA standard 335.

[10] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proc. HotOS-XI*, May 2007.

[11] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th SOSP*, 1995.

[12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, October 2004.

[13] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *Proc. HotOS-X*, June 2005.

[14] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *SIGOPS Oper. Syst. Rev.*, 40(1):95–99, 2006.

[15] IEEE. *Std 1003.1-1988 (POSIX)*, 1988.

[16] L4Ka Team, System Architecture Group, Dept. of Computer Science, Universität Karlsruhe. *L4 eXperimental Kernel Reference Manual, revision 5*, June 2004.

[17] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[18] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[19] J. Liedtke. Improving IPC by kernel design. In *Proc. 14th SOSP*, 1993.

[20] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Prentice-Hall PTR, 2nd edition, 1999.

[21] R. Pike. System Software Research is Irrelevant. `http://www.cs.bell-labs.com/cm/cs/who/rob/utah2000.pdf`, February 2000. Available December 2006.

[22] S. Ratnasamy. Capturing complexity in networked systems design: The case for improved metrics. In *Proc. 5th Workshop on Hot Topics in Networks (HotNets-V)*, November 2006.

[23] V. Sundaram, A. Chandra, P. Goyal, and P. Shenoy. Application performance in the QLinux multimedia operating system. In *Proc. 8th ACM Conference on Multimedia*, November 2000.

[24] VMware, Inc. `vmi_spec.txt`: Paravirtualization API Version 2.5. `http://www.vmware.com/pdf/vmi_specs.pdf`, February 2006.

[25] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proc. 20th SOSP*, 2005.

Relaxed Determinism: Making Redundant Execution on Multiprocessors Practical

Jesse Pool Ian Sin Kwok Wong David Lie
Department of Electrical and Computer Engineering
University of Toronto
{*pool,iansin,lie*}*@eecg.toronto.edu*

Abstract

Given that the majority of future processors will contain an abundance of execution cores, redundant execution can offer a promising method for increasing the availability and resilience against intrusions of computing systems. However, redundant execution systems rely on the premise that when external input is duplicated identically to a set of replicas executing the same program, the replicas will produce identical outputs unless they are compromised or experience an error. Unfortunately, threaded applications exhibit non-determinism that breaks this premise and current redundant execution systems are unable to account for this non-determinism, especially on multiprocessors. In this paper, we introduce a method called *relaxed determinism* that is utilized by our system, called *Replicant*, to support redundant execution with reasonable performance while tolerating non-determinism.

1 Introduction

Recent trends in computing hardware indicate that the vast majority of future computers will contain multiple processing cores on a single die. By the end of 2007, Intel expects to be shipping multi-core chips on 90% of its performance desktop and mobile processors and 100% of its server processors [6]. These multiprocessors can offer increased performance through parallel execution, as well as added system reliability and security through redundant execution.

Redundant execution is conceptually straightforward. A redundant execution system runs several *replicas* of an application simultaneously and provides each replica with identical inputs from the underlying operating system (OS). The redundant execution system then compares the outputs of each replica, relying on the premise that their execution is solely determined by their inputs, such that any divergence in their outputs must indicate a problem. For example, executing identical replicas has been used to detect and mitigate soft-errors [2]. More recently, there have also been several proposals to execute slightly different replicas to detect security compromises [4] and leaks of private information [9]. However, none of these systems have addressed threaded workloads.

Systems that support redundant execution via the OS kernel or a virtual machine monitor cannot account for the non-determinism that occurs among replicas when running threaded workloads on multiprocessors [3, 4, 9]. For similar reasons, systems that support deterministic replay also fall short on multiprocessor systems [5, 8]. This non-determinism undermines redundant execution by causing *spurious divergences* among replicas that are not the result of any failure or attack.

We believe the key insight that will allow efficient support for redundant execution on multiprocessors is that, in many cases, differences in the order that events are delivered to an application will not result in significant differences in application behavior. In light of this, we propose a relaxed deterministic execution model that is analogous to the relaxed memory consistency models used to improve performance on modern processors [1]. Modern processors relax the memory ordering guarantees they provide, and only enforce strict ordering when the software developer uses a memory "fence" or "barrier" instruction. Similarly, our system loosely replicates the order of events among replicas, and only enforces a precise ordering when instructed to do so through *determinism hints* that the developer inserts into the application. With the right relaxed determinism model, redundant execution on multiprocessors can be supported with reasonable overhead on existing processors.

We have implemented a redundant execution system that supports relaxed determinism, called *Replicant*, which is able to both increase the availability of a system, as well as prevent intrusions. In this work, we will begin with a detailed description of the problem faced

```
 1: int counter = 0;
 2: void thread_start(){
 3:   int local;
 4:   lock();
 5:   counter = counter + thread_id();
 6:   local = counter;
 7:   unlock();
 8:   printf("%d\n",local);
 9: }
10: void main(){
11:   thread_create(thread_start); // thread id = 1
12:   thread_create(thread_start); // thread id = 2
13:   thread_create(thread_start); // thread id = 3
14: }
```

Replica 1:	**Replica 2:**
Thread 1 prints **"1"**	**Thread 2** prints **"2"**
Thread 3 prints **"4"**	**Thread 3** prints **"5"**
Thread 2 prints **"6"**	**Thread 1** prints **"6"**

Figure 1: Code example illustrating non-determinism in a threaded program. Not only can the order of the thread outputs between Replica 1 and Replica 2 differ, but the contents of the outputs may differ as well.

by redundant execution systems on multiprocessors and follow with our approach to relax deterministic replication. We then give high-level details on annotating applications for Replicant, as well as some preliminary results.

2 Problem Description

Redundant execution systems rely on the presumption that if inputs are copied faithfully to all replicas, any divergence in behavior among replicas must be due to undesirable behavior, such as a transient error or a malicious attack. On such systems, the replication of inputs and comparison of outputs are typically done in the OS kernel, which can easily interpose between an application and the external world, such as the user or another application on the system. However, since inter-thread communication through shared memory is invisible to the kernel, and relative thread execution rates on different processors are non-deterministic, events among concurrent threads in a program cannot be replicated precisely and efficiently, leading to spurious divergences.

To illustrate, consider the scenario described in Figure 1. Three threads each add their thread ID to a shared variable, `counter`, make a local copy of the variable in `local`, and then print out the local copy. However, as illustrated below the program, the threads may update and print the counter in a non-deterministic order between the two replicas. In Replica 1, the threads print "1", "4" and "6" because they execute the locked section in the order (1, 3, 2) by thread ID. On the other hand, the threads in Replica 2 print "2", "5" and "6" because they execute the locked section in order (2, 3, 1). This example demonstrates that threaded applications may non-deterministically generate outputs in both different orders and with different values.

To avoid these spurious divergences, the redundant execution system must ensure that the ordering of updates to the counter is the same between the two replicas. If the redundant execution system ensures that threads enter the locked region in the same order in both replicas, then both replicas will produce the same outputs, though possibly in different orders. If the system further forces the replicas to also execute the `printf` in the same order, then both the values and order of the outputs will be identical.

A simple solution might be to make accesses to shared memory visible to the OS kernel, by configuring the hardware processor's memory management unit (MMU) to trap on every access to a shared memory region. For example, since `counter` is a shared variable, we would configure the MMU to trap on every access to the page where `counter` is located. However, trapping on every shared memory access would be very detrimental to performance, and the coarse granularity of a hardware page would cause unnecessary traps when unrelated variables stored on the same page as `counter` are accessed.

A more sophisticated method is to replicate the delivery of timer interrupts to make scheduling identical on all replicas. While communication through memory is still invisible to the kernel, duplicating the scheduling among replicas means that their respective threads will access the counter variable in the same order, thus resulting in the exact same outputs. Replicating the timing of interrupts is what allows systems like ReVirt [5] and Flashback [8] to deterministically replay threaded workloads. Unfortunately, as the authors of those systems point out, this mechanism only works when all threads are scheduled on a single physical processor and does not enable replay on a multiprocessor system. This is because threads execute at arbitrary rates relative to each other on a multiprocessor and as a result, there is no way to guarantee that all threads will be in the same state when an event recorded in one replica is replayed on another.

Finally, a heavy-handed solution might be to implement hardware support that enforces instruction-level lock-stepping of threads across all processors. Unfortunately, this goes against one of the primary motivations for having multiple cores, which is to reduce the amount of global on-chip communication. In addition, it reduces the opportunities for concurrency among cores, resulting in an unacceptably high cost to performance. To illustrate, a stall due to a cache miss or a branch misprediction on one core will also stall all the other cores in a replica.

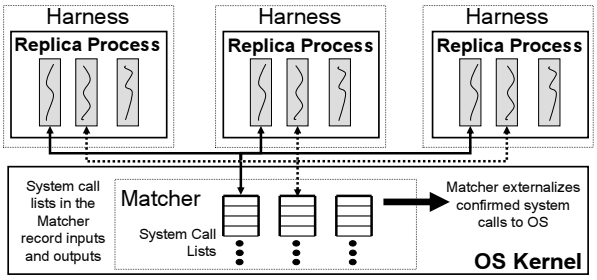


Figure 2: Replicant Architecture.

In summary, to support threaded applications on a multi-core architecture, the redundant execution system must be able to handle outputs produced in non-deterministically different orders among replicas. The redundant execution system must also be able to deal with the non-deterministic ordering of communication among replicas, which may result in divergent replica output values. In both cases, the system must either enforce the necessary determinism at the cost of some lost concurrency, or it must find ways to tolerate the non-determinism without mistaking it for a violation.

3 Approach

Due to the non-deterministic ordering of communication events, threaded applications can produce outputs in different orders and of different values when given the same inputs over consecutive runs. Unfortunately, forcing deterministic execution for replication by lock-stepping or trapping on shared memory accesses will have a high performance cost. Instead, Replicant allows the ordering of events to diverge between replicas, and requires determinism hints from the developer to indicate when a divergence in ordering can result in a divergence in output values. In this section, we will describe Replicant's system model, enumerate the relaxations to determinism, and the primitives Replicant provides to remove these relaxations.

3.1 System Model

Replicant implements an input replicating and output matching architecture that is tolerant to the reordering of events. Like other redundant execution systems, Replicant manages the inputs and outputs of several replicas of an application to appear as a single process to an external observer. However, unlike other systems, Replicant allows replicas to execute independently and does not assume that divergence in behavior is necessarily indicative of an application problem. Instead, Replicant buffers outputs and only makes them *externally visible* when they are *confirmed* (i.e. independently reproduced) by the majority of replicas. This makes Replicant tolerant to the non-deterministic reordering of outputs that concurrent applications typically exhibit.

Conceptually, Replicant can be viewed as computing the outputs of a correct execution from the system calls that the majority of replicas make. While an adversary may be able to compromise a subset of replicas, a majority is needed to subvert the externally visible behavior of the application. By increasing the number of replicas and introducing differences among replicas, such as address space randomization, we can make it arbitrarily improbable that an adversary will be able to simultaneously compromise enough replicas with the same attack. Replicant can also improve the availability of a system by removing any crashed or unresponsive replicas, thus allowing the remaining replicas to carry on execution.

Replicant's architecture is described in Figure 2. Replicant strives to increase performance by removing dependencies between replicas. To do this, Replicant should allow replicas to execute independently and diverge in their behavior. This is achieved by executing each replica in an OS sandbox, called a *harness*, which includes a private copy of the process-specific OS state and a copy-on-write file system. The purpose of the harness is to replicate the underlying OS state with enough fidelity such that the replicas are not aware that their outputs are actually being buffered (e.g. a replica will never notice an unconfirmed write to the file system). Harness state is visible only to the replica itself and is kept up-to-date by applying the outputs and effects of system calls made by the replica to the harness. Replicant also adds a *matcher* component to the OS kernel for each set of replicas. The purpose of the matcher is to fetch and replicate inputs from the external world into the harness, and determine when outputs from the harness should be made externally visible. The matcher is implemented as a set of system call lists that buffer the arguments and results of system calls made by the replicas. System calls are matched based on the thread identifier, the system call name and its arguments.

As summarized in Table 1, Replicant splits the handling of each system call invoked by a replica between the replica's harness and the matcher depending on whether the system call requires inputs or creates outputs, and whether those inputs and outputs are external or not. A non-external input is one that can be derived from the harness state, such as a `read` from a file on the copy-on-write file system, while an external input is one that must be derived from the OS, such as a `read` from the network or from a device. Replicant records the inputs from external system calls because they may not yield the same inputs if performed again at a later

	Does not Require External Input	Requires External Input
Does not have Externally Visible Output	Execute within harness.	**If system call matches a list entry:** Replay recorded inputs to the harness. **If system call does not match any list entries:** Execute system call on OS and record system call in the list.
Has Externally Visible Output	Execute system call within harness and buffer the output in the matcher until confirmed.	Extrapolate the result based on current OS state and return it to the harness. Defer execution on OS until the system call is confirmed by the matcher.

Table 1: Replicant's handling of system calls.

time. Replicant then replays the recorded inputs to other replicas when they make the same system call. System calls that do not require external input do not need to be buffered because each replica is initially provided with identical copies of the OS state in their harness.

Similarly, a non-external output is one that another application or user on the system cannot perceive, such as a `write` to a pipe between two threads in a replica, and an external output is one whose effects are externally visible, such as an `unlink` that deletes a file. Output system calls are executed on the harness, and if they are externally visible, they are committed to the external OS state when confirmed. For example, a `write` to the file system is a system call with external output. As such, its output is applied to the harness and committed when confirmed – which will succeed unless there is a catastrophic failure of the disk. However, a `write` to a socket is a system call with external output but also requires external input derived from the matcher's socket as opposed to the harness. Since the system call cannot be executed until confirmed, Replicant extrapolates the external input from the state of the socket and allows the replica to proceed. The socket `write` is buffered and externalized when confirmed.

3.2 Determinism Hints

Similar to relaxed memory consistency models, Replicant's behavior can be modeled as a set of relaxations that are made to the strict determinism that would be enforced by lock-stepping replica execution. From our system model, we can enumerate those relaxations and provide two determinism hints with which the programmer can temporarily suppress those relaxations when needed.

First, because Replicant does not explicitly force a deterministic ordering of events among replicas, they may produce divergent outputs due to non-determinism as illustrated in Figure 1. Since Replicant will not externalize divergent outputs, we provide the developer with a *sequential region* hint to enforce a deterministic ordering on events that affect external outputs. A sequential region encompasses a section of code, and Replicant ensures that all threads in all replicas will pass through sequential regions in the same order. This concept is similar to the shared object abstraction introduced by LeBlanc et al. [7].

Sections of code that contain inter-thread communication that can affect external outputs must be executed within sequential regions. Inserting sequential regions is straightforward in a conservatively written application where all accesses to shared memory are protected by locks. Because threads communicate through the locked memory, a simple solution is to place a sequential region around each critical section. By ensuring that memory accesses in sequential regions occur in the same order on all replicas, Replicant provides the same guarantees as lock-stepping but at a lower cost since deterministic execution is only enforced when threads are accessing shared data structures.

The second source of non-determinism is caused by instances where Replicant is unable to extrapolate the results of a system call that has an external output and requires an external input. A concrete example of this occurs when an application writes to a network socket. Replicant will return a result based on the state of the socket at the time the replica makes the system call. Later, when the system call is confirmed and is executed on the OS, the remote side may have closed the connection causing the socket write to fail. Other replicas who made the system call earlier may have been led to believe that the write succeeded because the remote client was still connected at the time. This leads to divergent results being returned to the replicas. Under these circumstances, the developer may suppress the relaxed determinism with a *synchronize syscall* hint. This hint instructs Replicant to cause the thread to block when it executes the next system call until it is confirmed, thus relieving Replicant of the need to extrapolate the return value.

Replicant also relaxes consistency between inputs and external outputs by delaying outputs until they are con-

firmed. Thus, Replicant may reorder inputs and outputs with respect to an earlier output in a way that is conceptually similar to the memory consistency guarantees provided by Partial Store Order [1]. The developer can also use the synchronize syscall hint to suppress this relaxation.

4 Annotating Applications

In the previous section, we enumerated the sources of non-determinism that may result in divergent outputs among replicas. Since Replicant will not externalize outputs with divergent values, the application developer must use determinism hints to annotate the events that can affect the outputs required for an external observer to perceive a correct application execution. Replicant will ensure that annotated events are deterministically replayed in all replicas.

Replicant introduces non-determinism among replicas in two ways: through non-deterministic ordering of inter-thread communication events and through non-deterministic input values extrapolated from system calls that have external output. Figure 1 illustrates inter-thread communication occurring through a shared counter variable. In this example, the developer can use a sequential region hint to ensure that the threads in both replicas access the shared variable in the same order, thus causing the outputs between the two replicas to be identical.

Figure 1 illustrates a key insight that makes it easier to place sequential region hints – accesses to shared variables are typically protected by critical sections defined by `lock` and `unlock` pairs. This has motivated us to create a sequential region programming interface which reflects that of locks. Replicant extends the Linux kernel with a `begin_seq` system call and an `end_seq` system call, which informs Replicant when a thread is entering and leaving a sequential region respectively. The developer uses these system calls by placing a `begin_seq` whenever a critical section begins, such as before the `lock` statement in Figure 1, and an `end_seq` whenever the critical section ends, such as right after the `unlock`. Sequential regions need only be inserted if the ordering of events can affect externally visible outputs. For example, if the program in Figure 1 did not print the intermediate values of the shared counter variable on line 8, but instead only printed the final value after all threads had updated it, then no sequential regions would be needed. This is because the thread ordering no longer has any effect on the application output. Since sequential regions enforce ordering across threads, they can reduce opportunities for concurrency, and should be used only when necessary.

In annotating an application, the developer may need to annotate several critical sections with sequential regions. To avoid adding unnecessary dependencies between critical sections protected by unrelated locks, Replicant allows the developer to define an arbitrary number of sequential region *domains*. Replicant enforces the order in which threads cross sequential regions that belong to the same domain, but does not enforce any order on sequential regions in different domains. As a result, there is a one-to-one mapping between locks in an application and sequential region domains, and each critical section that is protected by a certain lock maps to a sequential region in the corresponding domain. Sequential region domains are initialized through the `init_seq` system call, which takes a word-length domain identifier as an argument. This identifier is also passed as an argument to `begin_seq` and `end_seq` calls to identify which domain the sequential region belongs to.

While one can infer most of the inter-thread communication in an application from its use of locks, developers frequently find application-specific opportunities to increase performance by avoiding the use locks when accessing shared variables. As a result, when porting applications, we have found that while using information gleamed from the locks to automatically add sequential regions saves a great deal of time, some amount of manual analysis is usually required to discover the communication that does not occur in a critical section, but can still affect external output values. Qualitatively, we have found that inserting sequential regions is as difficult, and very similar in process, to inserting locks to parallelize an application. While the proper use of locks is certainly not trivial, they are in common use in concurrent applications today. Therefore, we feel that, if done at the time of development, the addition of sequential regions will not be an overly heavy burden on the application developer.

The other circumstance where Replicant may introduce non-determinism among replicas is by returning different extrapolated input values in response to system calls. In these cases, a synchronize syscall hint can be used to eliminate non-deterministic inputs at the developer's request. Similar to the sequential region, the developer need only insert this hint if the extrapolated input of the system call will affect the application's output values. We have added the `make_sync` system call for use in application annotations. This annotation is only required before instances of system calls with external inputs and extrapolated outputs where the application or a library checks the return value of the system call and takes some output action based on the return value.

5 Preliminary Results

We study the effectiveness of a 2-replica implementation of Replicant on six representative threaded appli-

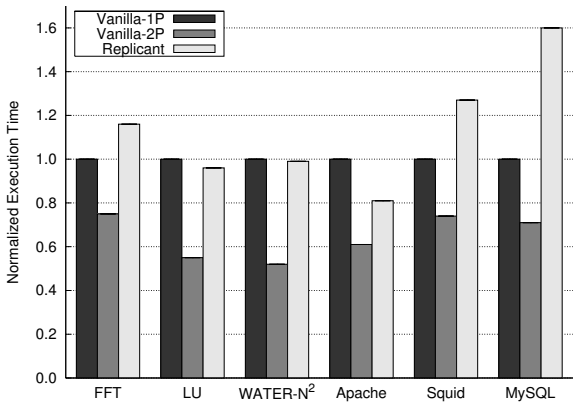


Figure 3: Performance of Replicant on six representative threaded applications as compared to 1P and 2P.

cations. We have ported three SPLASH-2 benchmarks (FFT, LU and WATER-N^2), Apache HTTP Server 2.2.3 (with worker Multi-Process Module), Squid Web Proxy Cache 2.3.STABLE9 (with asynchronous I/O enabled) and MySQL 5.0.25. All benchmarks were performed on an Intel Core 2 Duo 2.13GHz machine with 1GB of memory running Fedora Core 5 on a Gigabit network. The working set of all benchmarks fit in memory and the number of threads was increased until the dual processor vanilla benchmark could no longer utilize any more CPU time. We note that this does not mean that applications were necessarily able to utilize both CPUs to their maximum.

Figure 3 illustrates the performance of Replicant as compared to unmodified (vanilla) application performance on single processor and dual processor configurations. The comparison against the single processor performance is indicative of the case where the vanilla application is unable to make use of all processors available due to lack of sufficient parallelism. This is a reasonable scenario considering that future processors are projected to have many cores.

We find that there are three major application-dependent factors in Replicant performance. The first is how well the application balances load among threads. Squid has poor load balance and exhibits poor performance, while Apache, a very similar application, has good load balance and enjoys good performance. The second is the number of determinism hints that need to be invoked. MySQL has many sequential regions due to its frequent use of locks. Since sequential regions reduce opportunities for concurrency, MySQL experiences higher overhead. Finally, the ratio of user-space to kernel-space execution will affect application performance. Applications that spend much of their time in the kernel will experience less overhead because many kernel operations are only performed once, where as user space execution must always be duplicated.

6 Conclusion

By relaxing the determinism requirement among replicas in a redundant execution system, Replicant is able to provide better security and reliability at a lower cost to performance than systems that enforce strict deterministic replication of execution.

Acknowledgments

We would like to acknowledge Lionel Litty, Tom Hart, Ashvin Goel and the anonymous reviewers for their helpful comments. This work was funded in part by an NSERC Discovery Grant and a MITACS Seed Grant.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop advanced architecture. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN)*, pages 12–21, June 2005.

[3] A. L. Cox, K. Mohanram, and S. Rixner. Dependable $\neq$ unaffordable. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability*, pages 58–62, Oct. 2006.

[4] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-Variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, pages 105–120, Aug. 2006.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, Dec. 2002.

[6] Intel Corp., 2007. `http://www.intel.com/technology/magazine/computing/quad-core-1206.htm` (Last accessed: 03/08/2007).

[7] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, Apr. 1987.

[8] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 Annual Usenix Technical Conference*, pages 29–44, June 2004.

[9] A. Yumerefendi, B. Mickle, and L. Cox. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007.

Compatibility is Not Transparency: VMM Detection Myths and Realities

Tal Garfinkel, Keith Adams, Andrew Warfield, Jason Franklin
Stanford University,VMware,UBC/XenSource,Carnegie Mellon University

Abstract

Recent work on applications ranging from realistic honeypots to stealthier rootkits has speculated about building *transparent* VMMs – VMMs that are indistinguishable from native hardware, even to a dedicated adversary. We survey anomalies between real and virtual hardware and consider methods for detecting such anomalies, as well as possible countermeasures. We conclude that building a transparent VMM is fundamentally infeasible, as well as impractical from a performance and engineering standpoint.

1 Introduction

Recently there has been significant interest in providing virtual machine *transparency* – making virtual and native hardware indistinguishable under close scrutiny by a dedicated adversary, thus preventing VMM detection.

Interest in transparency stems from a variety of applications. Those exploring VMM-based worm detectors [9], malware detectors [18, 21] and honeypots [19, 8, 5] wish to disguise their VMs as normal hardware to avoid introducing an easy heuristic for detection and evasion. Anti-virus vendors are eager to cloak their use of VMs in identifying newly released exploits for similar reasons [23, 11]. Others have proposed offensive uses of virtualization in the form of VM-based rootkits (VMBRs) [10, 17, 24], hoping to leverage the transparency of VMMs to cloak their presence and provide an ideal attack platform [1]. We believe the transparency these proposals desire is not achievable today and will remain so.

The belief that VMM transparency is possible is based on a mistaken intuition that *compatibility* and *performance* imply *transparency* i.e. that once VMMs are able to run software for native hardware at native speeds, they can be made to look like native hardware under close inspection. This is simply not the case.

[1]Concern about virtualization-based rootkits has led to an advisory from Microsoft suggesting that CPU-based virtualization extensions be disabled in firmware on some systems [12].

While commodity VMMs conform to the PC architecture, virtual implementations of this architecture differ substantially from physical implementations. These differences are not incidental: performance demands and practical engineering limitations necessitate divergences (sometimes radical ones) from native hardware, both in semantics and performance. Consequently, we believe the potential for preventing VMM detection under close scrutiny is illusory – and fundamentally in conflict with the technical limitations of virtualized platforms.

Previous discussions of VMM transparency have generally failed to acknowledge the breadth and depth of virtualization induced anomalies, instead narrowly focusing on a few simple techniques for detection [11]. The conflict between transparency and practical demands such as performance and ease of implementation have also been largely overlooked. Optimistic hopes that emerging hardware support will mitigate these issues is misplaced.

Our thesis is that preventing VMM detection in the face of a dedicated adversary is generally impractical. In the next section, we survey the many types of disparities between native and virtual hardware, and means of detection. We then discuss limitations associated with trying to ameliorate these differences. We end on a positive note, observing that while infeasible, VMM transparency is likely of marginal benefit to defenders in the foreseeable future as virtualization becomes ubiquitous.

2 Virtualization Anomalies

We begin with a taxonomy of virtualization induced anomalies and detection methods. Our discussion does not aim to be exhaustive, instead we hope to convey the scope of the VMM transparency problem, and the obstacles to its solution.

In our threat model the VMM *passively* avoids detection. To succeed, virtual hardware must be sufficiently similar to physical hardware to be indistinguishable to an adversary in the guest OS. We explicitly ignore *active* attempts at thwarting detection, through modifying guest

code to disable detectors, on the grounds that such active approaches rely on *a priori* knowledge of the detector, making them inapplicable to the detection of novel—a.k.a. "zero-day"—detectors.

Our discussion makes no distinction between anomalies visible at kernel vs. user level, as we assume the reader can infer these by context. Further, this is often a false dichtomoy, as timing and logical semantics of virtual hardware are often as evident at user-level as they are in the kernel.

2.1 Logical Discrepancies

CPU Discrepancies. Logical discrepancies are semantic differences in the interfaces of real and virtual hardware. Most current VMM detection methods exploit differences in the virtual CPU interface of VMMs such as VMware Player or Microsoft VirtualPC that violate *x*86 architecture. Inaccuracies in the execution of some non-virtualizable instructions, such as *SIDT*, *SGDT*, and *SSL*, allow user-level inspection of privileged state [15].

Since these and other discrepancies are unimportant to the vast majority of software, VMMs make no effort to hide them. Intel's VT and AMD's SVM eliminate many obvious discrepancies, leading to speculation that hardware virtualization makes great strides towards providing transparency. This view is mistaken for several reasons.

As with software VMMs, transparency is secondary to providing an efficient and compatible execution environment. Thus, architecturally visible differences exist in current CPU virtualization support, allowing straightforward detection of hardware-assisted hypervisors [2]. Paravirtual VMMs intentionally extend the CPU in non-transparent ways. Xen and Denali provide modified software MMU architectures to ease implementation and improve performance [22, 6] and commercial VMMs have embraced a similar approach [4]. Even with compatible and performant hardware virtualization, this approach offers benefits, and thus will likely remain common.

Off-chip Discrepancies. SVM and VT only address CPU virtualization. Off-CPU differences between physical and virtual hardware abound, both for reasons of engineering ease and I/O performance. For example, modern chipsets are difficult to model [3]. For simplicity, the VMware virtual platform always emulates an *i*440bx chipset, leading to absurd hardware configurations: two AMD Opteron CPUs and 8 GB of RAM in an Intel motherboard from the Clinton administration, for instance. Operating systems run in such bizarre environments, because the firmware layer makes OS scrutiny of the chipset unnecessary.

I/O paravirtualization is another source of strange looking hardware. VMware, for example, provides various network, SCSI, and video cards that look nothing like any physical device, have PCI device and vendor ID's specific to VMware, and require their own device drivers. A detector can easily flag such non-hardware as proof of a VMM.

The VMM could try to emulate a richer set of devices, allowing more plausible hardware configurations. However, writing and maintaining software models of modern devices is difficult and costly. The VMM implementer further risks introducing new opportunities for detection in the form of bugs or omissions in emulated devices. Maintaining a comprehensive library of such emulated devices is a mammoth task that would require continuous engineering effort as new devices become available.

The VMM may also present guest hardware via *device pass through*, allowing guests to program physical devices directly. While this approach appears to improve transparency, it introduces two major problems: First, the VMM must audit device interactions to prevent DMA-based access to arbitrary memory; this effectively requires a complete emulated device model in order to parse device requests. Second, such pass through devices may no longer be shared with other VMs, defeating one of the basic purposes of virtualization.

Hardware support for virtualization is progressing to allow VMM protection from DMA through IOMMUs [3]. Current IOMMU proposals do not provide restartable semantics for DMA faults, however. Thus, even in the presence of an IOMMU, resource-efficient pass through will require virtualization-specific device interfaces [14], defeating any transparency benefits of device pass through.

2.2 Resource Discrepancies

VMMs share physical resources with their guests, including CPU cycles, physical memory, and cache footprint. To survive physical reboots, persistent storage is also required. Irregularities in the availability of these resources can betray the presence of a VMM.

Consider the TLB. VMM and guest virtual address mappings both compete for the same small pool of TLB entries. Software that is sensitive to TLB pressure can detect VMM use of this shared resource.

To demonstrate the feasibility of this approach, we constructed a simple TLB-sizing utility which changes page table entries (PTEs) *without* executing a TLB-

[2] For example, native *x*86 CPUs block non-maskable interrupts (NMIs) after delivery of an NMI until execution of the *IRET* instruction, but VT hardware does not provide a corresponding "block NMIs" bit [7]. Similarly, native *x*86 CPUs hold off debug exceptions for a one-instruction window following *M*OV %SS instructions. AMD's SVM provides no information about pending debug exceptions if an exit occurs in such a window [2]. We constructed a simple SVM detector based on this discrepancy in less than 100 lines of C and assembly.

[3] Modern chipsets are complex and rapidly evolving. Beyond changes in basic bus and memory controllers, richer power managment and I/O, built in sound, video, and security functionality all contribute to this.

synchronizing instruction, and then performs loads through the altered PTEs. A load via the old mapping indicates a hit, while a load from the new mapping is the result of a miss. Our utility runs this TLB sizing algorithm twice; during the second run, we interleave the memory accesses with exiting operations that invoke the VMM. In the presence of a VMM, the second run computes a smaller TLB size, due to the entries consumed by the VMM.

We can imagine VMM counter-measures for this detection method. The VMM might model a hardware TLB of native size, using a partial shadow page table as the current "virtual TLB" contents. However, similar methods can detect VMM pressure on data and instruction caches, and could include replacement policy along with size as a detection criterion. In the limit, the VMM must run every guest memory access through a software cache simulator to avoid detection, incurring absurd performance overheads, and leaving the VMM even more vulnerable to detection by our next set of techniques.

2.3 Timing Discrepancies

Virtual and physical environments differ in their timing characteristics. These differences are not simply "virtualization overhead" that can be addressed by making hardware or software faster – they can manifest as variance in latency, relative differences in the latencies of *any* two operations, and the behavior of these latencies over time. Consider some examples.

Device virtualization is a rich source of timing anomalies. For example, a PCI device register that takes a hundred cycles to read on physical hardware might require only a single cycle when the virtual hardware register is in the processor cache – cache behavior may in turn cause the virtual access time to exhibit higher variance than a hardware access. In this case, it is not the operation's latency *per se* that betrays the presence of a VMM, but the run-to-run variance in latency.

Memory virtualization also has guest-visible performance consequences [20]. VMMs use page protection for local and global resource management, memory-mapped I/O (MMIO) emulation, and protection of the VMM itself. Guest accesses to VMM-protected pages often induce *hidden page faults*, eliciting a performance discrepancy that can span three decimal orders of magnitude. While some of these memory overheads will be eliminated by future hardware developments such as AMD's NPT and Intel's EPT technologies [13], resource management and MMIO overheads will remain regardless of the hardware mechanism used to effect them.

Virtualization of privileged instructions is a well-understood source of timing discrepancies, due to the hardware and software overheads of a round-trip to the VMM. While hardware virtualization support shrinks some of these overheads, it bloats others [1]. On the balance, hardware virtualization makes timing-based detection no less feasible; while it changes the timing fingerprint of the virtual environment relative to a software-only approach, the fingerprint still differs from that of native execution.

Detectors can use a variety of time sources:

Local Time Sources. Timers and periodic interrupt sources provide the simplest mechanisms for measuring latencies. Examples include the hardware timestamp counter (*rdtsc*), PIT, ACPI timer and local APIC timer.

Guests can also use a host of relative time sources. Most hardware betrays timing information implicitly, in that hardware operations have a predictable average latency. Guest software can use these latencies as a time source by racing events against one another.

For example, we might construct a race between the innocuous *NOP* instruction and the virtualization-sensitive *CPUID* instruction. Thread A repeatedly spends its entire quantum executing *NOP*s and incrementing a count of completed operations, while thread B, bound to a separate logical processor, does the same with *CPUID*s. Since thread A takes no exits, the VMM has no opportunity to throttle the rate of *NOP*s. Over time, an observing thread will see the ratio of *NOP*s to *CPUID*s converging on a higher value in a virtual environment than in a physical one.

In this example we have assumed the presence of multiple CPUs, but this race construction technique can use any source of concurrency. DMA transfers, interrupt latencies, the memory hierarchy, and OS scheduler activity, for example, all provide possible sources of timing data for detection.

Remote Time Sources. A VM can use nearly any communication from the outside world as a clock. Possible remote time sources range from overt use of the NTP protocol to covert timing channels, such as subtle variations in inter-packet arrival times. Any communication with an outside entity will provide access to a covert channel, guaranteeing access to covert clocks – a VM can then measure performance discrepancies with a high integrity remote time source that is difficult to detect or modify even in the presence of an active VMM warden.

A VMM can interpose on and modify local or remote time sources, in an attempt to undermine timing based detection. This alone does nothing to prevent timing attacks. Any alterations a VMM makes must still "look real" relative to other time sources, thus necessitating the use of extreme measures like simulation. Approaches like adding randomness to either local or remote timing channels [5] merely introduces another source of anomalies for measurement.

3 Trade-offs and Implications

We've considered a number of VMM detection strategies, and have found the options for foiling these strategies limited. While our list is incomplete, the ease of imagining new detection methods suggests that VMM transparency is difficult to the point of impracticality. We now take a step back to look at transparency from the VMM implementor's point of view, and find an inherent tension between VMM performance and transparency. We then consider the ramifications of a world without transparent VMs.

Practical Emulation Overhead. VMMs may attempt to hide timing discrepancies by manipulating local time sources, a technique known as *time dilation*. Time dilation can allow virtual time to progress more slowly than real time, buying the VMM time to conceal its overheads.

To achieve transparency, time dilation must account for all latencies appropriately. In the limit, modeling all absolute and relative time sources reduces to cycle-accurate simulation of the entire system, entailing extreme engineering hardship and huge runtime overhead. Commercial emulators that provide a logical view of hardware without cycle-accuracy, such as AMD's SimNow or VirtuTech's Simics, incur a performance penalty on the order of 10x. Cycle accurate simulators such as SimOS incur overheads on the order of 25,000x [16] to provide full accuracy.

Perhaps some VMMs would accept such performance in the name of stealth, but this tactic would be self-limiting. If the practice became common, attackers could simply rule out attacking machines whose performance is below that of desktop machines of 10 to 20 years ago; we propose that the number of valuable targets running on such meager hardware is vanishingly small.

Even if dilation of local time sources were practical, the entire effort would be undermined as soon as a network is attached due to the intractable problem of remote timing channels.

Relative Accuracy vs. Absolute Cost. Time dilation, and other heroic efforts at preventing local timing detection, ironically make the hypervisor *more* vulnerable to remote detection; as the hypervisor spends more absolute time emulating local hardware, the gap between local and remote time sources widens.

In general, accurate simulation of any one component (e.g. memory, peripherals, processor attributes) provides greater leverage for detection attempts based on relative time differences. Carried to its logical conclusion, the cycle-accurate simulation approach is all-or-nothing: if any guest operation escapes from the cycle-accurate prison, all of the VMM's efforts at cycle-accuracy only make the VMM's presence more obvious. Since we've argued that trapping all remote time sources is intractable, the cycle-accurate simulation approach is a dead end for systems with network connectivity.

Real System Performance. Users are primarily concerned with the *utility* of their VMs: that VMs run correctly, are performant, and use minimal system resources. VMM implementors sweat blood over speed. To achieve high performance, commercial VMMs make heavy use of the systems design principle of optimizing for the common case.

VMs diverging from the common case may encounter performance cliffs. For detection software, such cliffs are dead giveaways that a VMM is present, but for VMM implementors, they are simply the consequence of a trade-off that allows common cases to run fast. Stealth and performance are thus, to some extent, in inevitable conflict.

For example, VMMs that use shadow paging manage a cache of *shadow page tables*. The VMM derives a shadow page table for each guest page table, allowing efficient virtual address mappings for the corresponding guest virtual address space. This cache has a finite capacity, however. By using an enormous number of page tables, the guest can conduct an exhaustion attack on the shadow page cache, leading to high rates of hidden page faults and an easily detected performance cliff.

Other cliffs result from the VMM's dynamic adaptations to guest behavior. VMware's VMM, for example, uses a cache of binary translations of guest kernel code, enforcing coherency via page protection of the source binary [1]. These translations evolve over time, e.g. by producing special translations for accesses to memory-mapped I/O devices.

In typical guests, this technique performs well. However, its performance rests on many assumptions, e.g., that self-modifying code is rare, and that the past behavior of an instruction predicts its future behavior. A guest can violate these assumptions to cause an easily observed performance degradation. Even with hardware virtualization extensions, this problem remains, as binary translation is a useful technique for avoiding hot spots causing frequent (expensive) `VMEXIT`s [1].

These cliffs do not reflect a weakness in the construction of VMMs, but rather one in the assumption of transparency. By attacking the mechanisms that a VMM uses to provide high performance for normal guests, a diabolical guest will always be able to elicit worse performance from the system.

Don't worry – Be Happy. Those relying on VMM-based systems for monitoring have expressed concern over VMM detection while posing a variety of incomplete solutions. We believe these concerns are largely unwarranted. Virtualization has made massive inroads into enterprise data centers, a trend that is expected to con-

tinue. Soon, malware that limits itself to non-virtualized platforms will be passing up a large percentage of commercial, military and institutional targets. To the degree that malware disables itself in the presence of VMs, VMs become even more attractive for production systems. In the long run, malware authors are motivated to operate regardless of the presence of a VMM.

Whither VMBRs? The question, "Might VM-based rootkits (VMBRs) be useful if they were small, clever, hardware-accelerated, etc., enough?" no doubt remains in some readers' minds. We think not. No matter how minimal the hostile VMM is, it must consume physical resources, perturb timings, and take measures to protect itself from the guest, leaving it no less susceptible to detection than other VMMs. Further, one of King et al.'s primary motivations for introducing VMBRs was to provide a simpler environment to build malware than found in current kernel based rootkits [10]. Highly resource-constrained VMBRs would defeat this purpose.

Even if VMBR detection were difficult, VMBR prevention is trivial and highly effective. As King et al. note, a stub VMM that refuses to load unsigned VMMs provides complete protection from VMBRs.

Perhaps the most concise argument against the utility of VMBRs is: "Why bother?" VMBRs change the malware defender's problem from a very difficult one (discovering whether the trusted computing base of a system has been compromised), to the much easier problem of detecting a VMM.

4 Conclusion

The compatibility VMMs provide *seems* just a small step away from transparency; intuition suggests that the tiny gap between native and virtual platforms must only be a small matter of programming, a dash of additional hardware support, etc., away from vanishing. We have challenged this view by surveying the wide range of dissimilarities between real and virtualized platforms, both on principle and using examples from today's VMMs. While tomorrow's VMMs will change, performance will remain paramount. Consequently, virtual and native hardware are likely to remain highly dissimilar, and thus amenable to discrimination.

5 Acknowledgements

This work was supported in part by TRUST (The Team for Research in Ubiquitous Secure Technology), which is supported by National Science Foundation (award CCF-0424422). Jason Franklin performed this research while on appointment as a U.S. Department of Homeland Security (DHS) Fellow under the DHS Scholarship and Fellowship Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Homeland Security. We are also very greatful for feedback and discussions with Ole Agesen, Mendel Rosenblum, Jim Chow, and Jim Mattson.

References

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[2] AMD. *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005.

[3] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*, Feb. 2006.

[4] Z. Amsden, D. Arai, D. Hecht, and P. Subrahmanyan. Paravirtualization API Version 2.5. `www.vmware.com/pdf/vmi_specs.pdf`.

[5] K. Asrigo, L. Litty, and D. Lie. Using VMM-based Sensors to Monitor Honeypots. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, June 2006.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[7] Intel Corporation. *Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture*, April 2005.

[8] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. In *Proceedings of 13th USENIX Security Symposium*, Aug. 2004.

[9] E. Jonsson, A. Valdes, and M. Almgren. HoneyStat: Local Worm Detection Using Honeypots. In *Proceedings of Seventh International Symposium on Recent Advances in Intrusion Detection*, Sept. 2004.

[10] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[11] T. Liston and E. Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection . `http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf`, July 2006.

[12] Microsoft. CPU Virtualization Extensions: Analysis of Rootkit Issues. `http://www.microsoft.com/whdc/system/platform/virtual/CPUVirtExt.mspx`. Windows Hardware Developer Central, October 20, 2006.

[13] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), Aug. 2006.

[14] PCI SIG. *PCI I/O Virtualization Specifications*.

[15] J. Robin and C. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, Aug. 2000.

[16] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995.

[17] J. Rutkowska. Subverting Vista Kernel for Fun and Profit. Presented at Black Hat USA, Aug. 2006.

[18] S. Sidiroglou, J. Ioannidis, A. D. Keromytis, and S. J. Stolfo. An Email Worm Vaccine Architecture. In *Proceedings of the First Information Security Practice and Experience Conference*, 2005.

[19] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Oct. 2005.

[20] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[21] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, Feb. 2006.

[22] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[23] L. Zeltser. Virtual Machine Detection in Malware via Commercial Tools. `http://isc.sans.org/diary.php?storyid=1871`. Handlers Diary, November 19, 2006.

[24] D. D. Zovi. Hardware Virtualization-Based Rootkits. Presented at Black Hat USA, Aug. 2006.

Don't settle for less than the best: use optimization to make decisions

Kimberly Keeton, Terence Kelly, Arif Merchant, Cipriano Santos, Janet Wiener and Xiaoyun Zhu
Hewlett-Packard Laboratories, Palo Alto, CA
firstname.lastname@hp.com

Dirk Beyer
M-Factor, San Mateo, CA
dirk.beyer@m-factor.com

Abstract

Many systems design, configuration, runtime and management decisions must be made from a large set of possible alternatives. Ad hoc heuristics have traditionally been used to make these decisions, but they provide no guarantees of solution quality. We argue that operations research-style optimization techniques should be used to solve these problems. We provide an overview of these techniques and where they are most effective, address common myths and fears about their use in making systems decisions, give several success stories and propose systems areas that could benefit from their application.

1 Introduction

Decision-making problems abound in systems research, including questions of resource provisioning [6, 16], resource allocation and scheduling [17, 20, 23, 24, 30, 31], system administration and management [8], and application and system design [3, 5, 15, 25]. These problems are characterized by a large space of potential solutions, with complex tradeoffs between system performance, availability, reliability, manageability and cost. Given the large solution space, it's hard to keep the alternatives straight, let alone find the best solution. Even when a "good enough" or merely feasible solution (i.e., one that meets constraints) is desired, it can be hard to find. Furthermore, getting the answer wrong can be costly (e.g., in the time to recover from a disaster or in the monetary expense of over-provisioning physical resources), so there is a strong incentive to choose wisely.

Traditionally, systems researchers have used ad hoc domain-specific heuristics to solve these decision-making problems. Unfortunately, heuristics don't provide the best solution, nor do they provide any bounds on how close their solution is to the best. Recently, a new paradigm has emerged, where systems decision problems are cast as formal optimization or constraint satisfaction problems, allowing the use of operations research (OR) solution techniques, from mathematical programming to meta-heuristics.

In this paper, we argue that the systems community needs to leverage the more principled approach of formal optimization to solve design, configuration, runtime and management decision-making problems [26]. By optimization, we mean first formally specifying the problem, and then using any of several techniques to solve it. Specifying the problem means explicitly defining the objective, the constraints on a valid solution, and how input parameter values impact the goodness of a candidate solution. Formulating these aspects of the decision-making problem forces us to understand the underlying problem and the tradeoffs that we're trying to capture. This knowledge is useful, whether the problem is ultimately solved by standard OR techniques or by domain-specific heuristics.

Standard OR solution techniques provide many benefits. In many cases, these techniques provide optimal answers, which means that researchers don't need to worry that a heuristic might perform poorly for an as-yet-unseen corner case. The speed of current desktop machines makes it possible to use these techniques on many problems that would have been intractable even ten years ago. OR techniques encourage a clean separation between the problem statement and the solution method. Furthermore, the availability of commercial off-the-shelf solvers means that we as systems researchers can focus on specifying the problem at hand, rather than worrying about how to solve it.

In the remainder of the paper, we provide an overview of popular OR techniques and debunk common myths preventing their usage in the systems community. We discuss when to choose an optimization technique instead of an ad hoc heuristic. Finally, we give several success stories where optimization has been applied to systems problems and list several areas that are ripe for optimization in the future.

2 Optimization techniques

The OR community presents a variety of techniques to find optimal and approximate solutions to decision problems. To use any of these techniques, the first step is to describe the problem formally: what decisions must be made, which alternatives are feasible, and what the "goodness" metric is for comparing solutions. Decisions might include which outgoing link to use to transmit a message in a wireless routing environment, or whether to allocate a server to workload A or workload B. Constraints on alternatives may be either hard constraints, which cannot be violated, or soft constraints, where violations of the constraint incur penalties. The specific formalism varies with the technique, as described in the rest of this section. One common thread, however, is that all tradeoffs must be expressed in the same currency, such as execution time, throughput, monetary cost, or a "utility" composed from such metrics.

2.1 Techniques to find optimal solutions

Techniques such as mathematical programming (MP) provide an optimal answer to a decision problem. Among the approaches we consider, MP requires the most detailed knowledge of the decision problem. In an MP formulation, *decision variables* correspond to the choices to be made, *objective functions* quantify a candidate solution's "goodness," and *constraints* describe which solutions are feasible [11, 29]. The solver then determines an optimal solution, the top N solutions, a solution within $x\%$ of optimal, the best solution possible within a time budget, or simply a feasible solution.

Mixed integer programs (MIPs) are math programs with linear objective functions, constraints defined by linear inequalities, and decision variables that take on continuous or discrete (often binary) values. MIPs where all decision variables have continuous values are called linear programs (LPs). MIPs are appropriate for problems characterized by contention for additive resources and additive measures of system goodness.

2.2 Techniques to find feasible solutions

The OR community also provides techniques for finding feasible solutions to decision problems, such as constraint programming [12]. Constraint programming (CP) is an appropriate technique when the constraints can only be expressed by rules — logical statements such as "if you choose option A, you must also choose B, C and D." A constraint satisfaction problem consists of decision variables, each with a domain of valid discrete values, and a set of constraints governing feasible solutions. A solution is a complete assignment of variables that meets all of the constraints. CP is predominantly used to find feasible, rather than optimal, solutions.

2.3 Techniques for approximate solutions

Meta-heuristics are algorithms for finding near-optimal solutions, which are inspired by naturally-occurring phenomena, such as genetic algorithms [21], simulated annealing, and auctions. In a genetic algorithm (GA), an individual represents a feasible solution, and the genes of the individual represent decision variables. The most "fit" individuals are selected for the next population based on a fitness function, which is roughly equivalent to the objective function in a MIP. These techniques require less detailed knowledge of a problem's structure because they rely on a procedure, or "oracle," to determine the feasibility and goodness of a candidate solution. This oracle can be an analytic model, a lookup table, or even a simulation. Meta-heuristics provide few, if any, guarantees on the optimality of their solutions. However, it is sometimes possible to state probabilistically how close to optimal a solution is.

3 Myths and realities

Although optimization techniques are powerful, systems researchers are often skeptical about applying them to solve decision problems. In this section, we refute several common myths about optimization.

Myth: A simple heuristic is "good enough." If an easy-to-implement and quick-to-run heuristic exists, why not use it? If the problem doesn't require an optimal solution, is formal optimization overkill?

Reality: If a simple heuristic provides "good enough" answers, then it may be the appropriate choice. The challenge lies in quantifying what "good enough" means and determining if a solution meets it. In many cases, it's hard to determine what "good enough" is, without knowing the best that can be achieved. Even if the goal is balancing tradeoffs between conflicting goals, rather than finding an optimal solution, we still need to understand the relative costs of the alternatives, so that we know whether the appropriate balance is achieved. Without a formal specification, it can be hard to estimate how close to optimal a solution lies; with ad hoc approaches, it can usually be determined only empirically. Techniques like math programming provide a systematic solution with bounds on how close that solution comes to the optimal one. Furthermore, even if optimization techniques work only for small problem instances, their results can be compared with those of domain-specific heuristics, to help understand the heuristic's behavior for larger instances.

Myth: Problem formulation takes too much time. Formulating problems is often challenging, and it requires both domain expertise and knowledge of the optimization technique. Employing ad hoc domain-specific heuristics doesn't generally require such up-front, interdisciplinary effort.

Reality: The hardest part of problem formulation is understanding the problem — its goals and tradeoffs, as well as how to capture the underlying system's behavior. This first step is required whether the ultimate solution is a standard optimization technique or a domain-specific heuristic. Unfortunately, in the latter case, the explicit formulation step is often ignored, and researchers end up gradually "discovering" aspects of the problem as they successively refine their heuristic. The effort in formulation is well-spent, because it's easier to adapt the formulation as the decision question or constraints change than to adapt an ad hoc heuristic.

Myth: Formulating the problem requires too many simplifying assumptions. If too many simplifications are made, then the decision is not realistic, and the resulting solution may be meaningless.

Reality: Our collective experience is that simplifications are problematic only when we try to force a problem into a particular framework (e.g., force non-linear behavior into an LP). If one technique doesn't work, we need to try another one, or to break the problem down so that different techniques can be used for different portions of the problem (e.g., a MIP for resource provisioning and a heuristic for resource scheduling). Ultimately, if no optimization technique works, the formal description is still useful for understanding the problem and developing an ad hoc domain-specific heuristic.

Myth: Optimization techniques are too slow. Standard optimization techniques take too long to be useful for runtime management decisions.

Reality: The execution times of these techniques are highly dependent on the size of the problem and its structure. (For instance, linear programs can be solved more efficiently than non-linear ones.) Execution times can be under a second. Given that many decisions will be in effect for days or months, many decision-making problems can tolerate the execution times of OR techniques.

Myth: Inaccurate input data may result in bad decisions. Variations in the input values may cause variations in optimal solutions.

Reality: Sensitivity to the input values is a characteristic of the problem domain, rather than the solution technique. If we can't estimate input values with high accuracy, for example, because they are estimates of business utility, then it's important to do a sensitivity analysis to understand how the optimal solution varies with different input values.

Myth: Optimal solutions may not be easy to support. Optimal solutions may use non-standard configurations for a large hardware or software system, which may be hard to maintain.

Reality: It's difficult to capture intangible goals such as "manageability" in an objective function. If they can't be represented quantitatively in the objective function, it may be possible to restrict the space of candidate solutions to only those that fit the intangible criteria. Another possibility is to present a family of possible solutions to the user, who can then choose one based on the intangible goals.

4 When should I use optimization?

Four criteria must be met for a math programming or constraint solver to be useful. These criteria are: desire for a better solution than an ad hoc heuristic can provide, enough knowledge of the decisions to be made to express them formally, accurate and available input data so that the solver can compare alternative solutions, and sufficient time to run the solver. If only the latter two are met, then meta-heuristics may be appropriate. Otherwise, an ad hoc heuristic may be the only choice. If the answers to all four questions are all yes, then using optimization is the best bet.

Does this problem need an optimal solution? The stronger the desire to find the best solution, the more worthwhile it is to employ math programming or constraint programming techniques. These approaches can find feasible or, in the case of math programming, optimal solutions. Meta-heuristics and domain-specific heuristics don't provide any optimality guarantees.

Can the decisions and constraints be modeled formally? Math programming is appropriate if the system constraints can be modeled as sets of inequalities. If constraints can be specified only in Boolean terms, then constraint programming is a better choice. If system behavior can only be understood through simulation or black-box measurement, perhaps because of complex interactions between components, then a meta-heuristic or domain-specific heuristic is most appropriate.

Is enough input data available? Is it accurate enough? For math programming and constraint programming, complete input data must be available to evaluate all possible alternatives; unavailable data must be estimated with reasonable accuracy. Meta-heuristics may be able to get by with partial input information, because they can incorporate new or changed data at each step of the search space exploration. If input data is arriving continuously or can't be accurately measured or modeled, then an ad hoc heuristic is easier to use.

Is there enough time to compute an optimal answer? For optimization to be effective, the time to make

a decision should be shorter than the time frame for revisiting the decision with new data. Storage-related and wide area distributed run-time management decisions require answers in under a second, and MIPs can sometimes provide answers in under a second, depending on the size of the problem and the set of constraints. Configuration and capacity planning decisions that will take hours to days to implement can tolerate much longer decision-making latencies, from minutes to hours. Current commercial MP solvers, such as ILOG's CPLEX solver [13], can solve LP problems with hundreds of thousands of variables in minutes.

Applying formal optimization techniques may not be worthwhile in all circumstances. Other approaches may be more effective if: 1) the decision has only a minor impact on the quality (e.g., the performance, availability, power or manageability) of the overall solution; 2) it's easy to enumerate and evaluate all of the alternatives; 3) the alternatives are roughly equivalent in cost and benefit; 4) a formal technique won't provide the answer quickly enough to be useful; 5) the solution quality or inputs are hard to quantify; or 6) it's easy to change the decision if the result is unsatisfactory.

5 Where and how to use optimization?

We believe that many systems decision-making problems should be solved by standard optimization techniques. These problems are complex (and thus not easily solvable by ad hoc methods), the solutions have long-lasting impact (thus permitting longer solution times and requiring good solutions), and in many cases, the system parameters can be measured to provide accurate inputs. We summarize three areas where these techniques have been successfully applied, provide references to additional example success stories, and enumerate several classes of problems where optimization will be useful in the future.

5.1 Data recovery scheduling

We have addressed the question of scheduling recovery operations in a dependable storage system after a failure using MIP, genetic algorithm and domain-specific heuristic formulations [17]. Dependable storage systems protect application data by making copies through backup, snapshot and replication techniques. After a failure, applications must decide which copy to use for recovery. Some alternatives (e.g., restoring from a backup) provide fast recovery with non-trivial loss of recent updates, while others (e.g., restoring from a remote replica across a low-bandwidth network) provide minimal data loss at a potentially higher recovery time.

Applications incur financial penalties due to downtime, recent data loss and vulnerability to subsequent failures. Our objective is to minimize these penalties. The inputs to the problem are a set of penalty rates (e.g., dollars per hour for outages) for each application, device resource capabilities, and a recovery graph describing alternate recovery paths for each workload, including their operations, resource requirements and precedence relationships. The techniques choose a recovery path for each workload and determine a schedule for the recovery operations. Constraints govern the choices that can be made: for each application, only a single recovery path can be chosen, and the chosen schedule must satisfy the precedence constraints specified in the input recovery graph. Constraints also govern resource usage: the sum of all resource demands for a given device must not exceed the capabilities for that device.

We began by formulating a MIP, which we solved using ILOG's CPLEX solver [13]. However, we found that the MIP implementation had limited scalability. Even so, the MIP formulation gave us greater insight into the recovery scheduling problem, which we applied to the design of the GA and the domain-specific heuristic. We also used the MIP to establish the optimal solution for small problem sizes. We were even able to define larger problems based on the smaller ones, where we could extrapolate the optimal solution for the larger problem size. We compared the solutions provided by the other techniques against the MIP's optimal solution.

5.2 Publish-subscribe system

Corona [23] is a publish-subscribe system that provides asynchronous update notifications to its subscribers. Users register URLs they're interested in, and the system asynchronously sends them updates about changes posted to the URL. Changes are detected through cooperative polling by multiple nodes that periodically check the same URL and share any detected updates. Using more nodes for a URL improves update performance, but increases network load. The precise tradeoff between performance and load depends on several factors, including the number of clients requesting a URL, the content size, the update frequency, etc. Corona resolves the tradeoff by treating the number of nodes per URL as an optimization problem. The authors define several different objectives: optimizing performance while limiting load; minimizing load while bounding update delay; and several other performance metrics that depend on both the update rate and update delay per URL. The resulting non-linear optimization problem is solved quickly through their decentralized Honeycomb optimizer. A distinct advantage of this approach is that all the optimization objectives can be achieved through a common technique, as opposed to an ad hoc approach that would have required a separate heuristic for each case.

5.3 Web cache management

Web caches reduce network traffic and downloading latency, and can affect the distribution of Web traffic over the network through cost-aware caching. Web cache replacement policies choose which documents to evict when the cache is full, and this decision problem can be addressed through an explicit objective function. For example, Cao and Irani use objective functions based on combinations of temporal locality, document size, and network latency [5]. Kelly *et al.* propose a cache replacement algorithm that allows users to define the value of cache hits and that strives to maximize aggregate user value [19]. Both approaches can postpone the definition of the objective function until run-time, rather than specifying it at design time. Allowing users or run-time conditions to define the objective has familiar systems analogs, such as the `qsort()` function in the C library, which accepts an arbitrary client-supplied comparison function. These precedents show that adopting an OR-style optimization approach doesn't require us to hard-wire an objective function into our optimization designs; the objective function can instead be a placeholder, to be supplied by users.

5.4 Decision problems ripe for optimization

Many systems decision-making problems beg the use of optimization. Here we outline several such areas, providing references to published work that applies optimization and articulating specific open questions.

Resource provisioning: Numerous issues arise in provisioning server, network and storage resources to meet service level objectives [2, 6, 7, 16, 22, 27, 28]. For example, how many servers, network links, and bytes of storage are needed for competing workloads to meet their performance goals? How much redundancy (and what kind) is needed to guarantee the desired levels of reliability and availability? How many devices can be turned on, while still meeting power and cooling budgets? If all machines are not in use, which ones should be turned off? When a cooling unit fails, which machines should be turned off so that the current workload is least impacted, but the room doesn't overheat?

Resource allocation and scheduling: Which servers and storage devices should be assigned to which workloads and for how long, to meet performance [31] or availability [15] goals? Other goals may include maximizing customer revenue, minimizing energy [30], or meeting scheduling deadlines [3]. In a sensor network [20], which sensors should be powered off, and how should messages be routed to minimize energy consumption?

System administration and management: Many questions arise in managing system administration changes [18], migrating data [10], and setting application configuration parameters [8]. For instance, when should servers be upgraded to minimize application performance impact? How should data be migrated to newer storage, given bandwidth and ordering constraints?

Application and system design: Interesting questions emerge in contexts such as cache management for distributed data [5, 9, 14], distributed data replication strategies [26], determining checkpoint intervals for long-running computations [4], and database design [1]. For instance, which data should be replicated in web servers or distributed hash tables (DHTs) to minimize access time, minimize write time, or meet reliability guarantees? What is the right tradeoff between storing intermediate results and repeating computations after a failure? Which indexes and materialized views will minimize query execution time for a given query workload?

Although initial work has been done in these areas, many opportunities remain. As systems grow increasingly complex, we expect that the list will grow.

6 Conclusions

As system complexity increases, the number of decisions to be made, as well as the number of potential choices, increases. The key to solving these problems is to thoroughly understand the questions they ask – what should be decided, what solutions are reasonable, how to compare the alternatives, and what's important for picking the most appropriate solution. By formally formulating decision problems, the researcher gains greater insight into the problem's tradeoffs, regardless of how the problem is finally solved.

Systems researchers shouldn't settle for less than the best answers to decision-making questions. We should apply the principled approach of operations research techniques like math programming, constraint programming and meta-heuristics to obtain the best solutions. Now that we've described when these techniques are most useful, we hope you'll consider using them for the decision-making problems you face.

References

[1] S. Agrawal et al. Automated selection of materialized views and indexes for SQL databases. *Proc. of VLDB*, pp. 496–505, Sept. 2000.

[2] G. Alvarez et al. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM TOCS*, 19(4):483–518, Nov. 2001.

[3] E. Anderson et al. Value-maximizing deadline scheduling and its application to animation rendering. *Proc. of SPAA*, pp. 299–308, July 2005.

[4] J. Bent et al. Explicit control in a batch-aware distributed file system. *Proc. of NSDI*, pp. 365–378, Mar. 2004.

[5] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. *Proc. of USITS*, pp. 193–206, Dec. 1997.

[6] J. Chase et al. Managing energy and server resources in hosting centers. *Proc. of SOSP*, pp. 103–116, Oct. 2001.

[7] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. *Proc. of Supercomputing*, pp. 1–11, Nov. 2002.

[8] Y. Diao et al. Generic, on-line optimization of multiple configuration parameters with application to a database server. *Proc. of DSOM*, pp. 3–15, Oct. 2003.

[9] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, June 1982.

[10] J. Hall et al. On algorithms for efficient data migration. *Proc. of SODA*, pp. 620–629, Jan. 2001.

[11] F. Hillier and G. Lieberman. *Introduction to operations research*. McGraw Hill, 7th edition, 2001.

[12] J. Hooker. *Logic based methods for optimization*. John Wiley and Sons, 2000.

[13] ILOG, Inc. *CPLEX 8.0 User's Manual*, July 2002. Available from http://www.ilog.com.

[14] S. Jamin et al. Constrained mirror placement on the Internet. *Proc. of INFOCOM*, pp. 31–41, Apr. 2001.

[15] J. Janakiraman, J. R. Santos, and Y. Turner. Automated system design for availability. *Proc. of DSN*, pp. 411–420, June 2004.

[16] K. Keeton et al. Designing for disasters. *Proc. of FAST*, pp. 59–72, Mar. 2004.

[17] K. Keeton et al. On the road to recovery: restoring data after disasters. *Proc. of EuroSys*, pp. 235–248, Apr. 2006.

[18] A. Keller et al. The CHAMPS system: Change management with planning and scheduling. *Proc. of NOMS*, Apr. 2004.

[19] T. Kelly et al. Biased replacement policies for web caches: Differential quality-of-service and aggregate user value. *Proc. of 4th Web Caching Workshop*, March-April 1999.

[20] G. Mainland et al. Decentralized, adaptive resource allocation for sensor networks. *Proc. of NSDI*, May 2005.

[21] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs*. Springer-Verlag, 3rd edition, 1996.

[22] J. Moore et al. Making scheduling "cool": Temperature-aware workload placement in data centers. *Proc. USENIX Technical Conf.*, pp. 61–75, Apr. 2005.

[23] V. Ramasubramanian et al. Corona: A high performance publish-subscribe system for the world wide web. *Proc. of NSDI*, pp. 15–28, May 2006.

[24] J. Rolia, A. Andrzejak, and M. Arlitt. Application placement in resource utilities. *Proc. of DSOM*, Oct. 2002.

[25] A. Sahai et al. Automated policy-based resource construction in utility computing environments. *Proc. of NOMS*, Apr. 2004.

[26] E. G. Sirer. Heuristics considered harmful: Using mathematical optimization for resource management in distributed systems. *IEEE Intelligent Systems*, pp. 55–57, Apr. 2006.

[27] S. Uttamchandani et al. Chameleon: A self-evolving, fully-adaptive resource arbitrator for storage systems. *Proc. of USENIX Technical Conf.*, pp. 75–88, Apr. 2005.

[28] J. Ward et al. Appia: automatic storage area network design. *Proc. of FAST*, pp. 203–217, Jan. 2002.

[29] L. Wolsey. *Integer programming*. John Wiley and Sons, 1998.

[30] Q. Zhu et al. Hibernator: Helping disk array sleep through the winter. *Proc. of SOSP*, pp. 177–190, Oct. 2005.

[31] X. Zhu et al. Resource assignment for large scale computing utilities using mathematical programming. Technical Report HPL-2003-243R1, Hewlett-Packard Labs, Feb. 2004.

Hyperspaces for Object Clustering and Approximate Matching in Peer-to-Peer Overlays

Bernard Wong Ýmir Vigfússon Emin Gün Sirer
Dept. of Computer Science, Cornell University, Ithaca, NY 14853
bwong@cs.cornell.edu ymir@cs.cornell.edu egs@cs.cornell.edu

Abstract

Existing distributed hash tables provide efficient mechanisms for storing and retrieving a data item based on an exact key, but are unsuitable when the search key is similar, but not identical, to the key used to store the data item. In this paper, we present a scalable and efficient peer-to-peer system with a new search primitive that can efficiently find the k data items with keys closest to the search key. The system works via a novel assignment of virtual coordinates to each object in a high-dimensional, synthetic space such that the proximity between two points in the coordinate space is correlated with the similarity between the strings that the points represent. We examine the feasibility of this approach for efficient, peer-to-peer search on inexact string keys, and show that the system provides a robust method to handle key perturbations that naturally occur in applications, such as file-sharing networks, where the query strings are provided by users.

1 INTRODUCTION

Modern P2P substrates do not provide support for efficiently locating objects whose keys are not known precisely. In settings where queries are based on terms provided by users, imprecision stemming from partial specifications of keywords, common variations of search terms and misspellings are unavoidable. For instance, approximately 20% of all Google queries for "Britney Spears" misspell the artist's name [2]. Efficiently routing a query to a set of objects whose keys are close[1] but not identical to the search key is a difficult problem known as "approximate match."

Even though peer-to-peer systems were initially motivated by file-sharing, modern P2P substrates do not provide efficient primitives for approximate matching. Unstructured peer-to-peer systems [1] provide a *search* primitive, which is based effectively on query broadcast[2]. Gnutella nodes receiving the search query match it against their database of known items using a fuzzy similarity metric to yield the approximate matches. Such broadcast-based approaches are inefficient as they may take up to N hops in the worst case where N is the number of hosts, and place a super-linear aggregate load on the network. In contrast, structured peer-to-peer systems [21, 23, 28, 19, 16, 12] provide an efficient *lookup* primitive that can typically locate a target within $\log N$ hops. While these systems provide strong worst-case bounds, the lookup operation does not permit approximate matching. Naive approaches to layer approximate matching on top of a DHT lookup, by inserting each object under all possible key variations and performing every query in parallel with all k variants of the search key, lead to highly inefficient solutions because k is typically on the order of a few hundred even for a moderate length movie title with only two permuted characters. Finally, systems that permit *range lookups* [6, 8] can perform a lookup within a range defined by numeric coordinates, but are difficult to adopt for use with approximate string matching. Overall, existing systems provide inefficient and approximate search or efficient and precise lookup, but not efficient and approximate match.

In this paper, we present e-llama, a scalable peer-to-peer system that can efficiently find the k closest data items for any search key. The central insight behind e-llama is to define a very high-dimensional space (a *hyperspace*) in which every object and node is assigned a virtual coordinate. The bases (axes) for the hyperspace consist of string labels. The virtual coordinate for every object is the tuple created by measuring the edit-distance to each of the axis labels. For instance, for bases **aaa**, **cbc** and **abd**, the keys **abc**, **abd** and **ddd** would map to the points $\langle 2, 1, 1\rangle$, $\langle 2, 2, 0\rangle$ and $\langle 3, 3, 2\rangle$, respectively. This virtual coordinate assignment captures the relative similarities of the strings

[1] Closeness here is defined based on an application-supplied similarity measure, such as edit distance. Our approach requires only that this measure σ obey $\sigma(a, b) \geq 0$, $\sigma(a, b) = \sigma(b, a)$ and $\sigma(a, c) \leq \sigma(a, b) + \sigma(b, c)$. We use edit-distance as a running example without loss of generality.

[2] Optimizations, such as supernodes and expanding ring search, make the broadcast process more efficient, but the primitives are still based fundamentally on flooding.

through the edit-distances to the axis labels.

An efficient algorithm, based on small-worlds [13], for navigating this multi-dimensional hyperspace enables e-llama to quickly identify approximately matching objects. E-llama assigns a random location in hyperspace to each overlay node, and each node maintains the set of objects for which it is the closest node. Every node also keeps track of other nodes in concentric rings of exponential radii. E-llama routes a query to the closest node for a target coordinate by greedily determining the peer in its rings that is closest to the target coordinate and forwarding the query. Each forward brings the query closer to the target coordinate and to a node with more information in the proximity of the targeted region than the previous node. This protocol converges to the closest node in $O(\log N)$ hops with high probability. Once the target node has been located, the search expands in a sphere around the target until k matching objects are found.

Overall, this paper makes three contributions. First, it describes a new technique for constructing a synthetic space in which similar keys are clustered. Second, it describes a scalable and efficient protocol for mapping this space to nodes and routing queries to nodes, yielding a DHT with an approximate match primitive. Finally, it demonstrates the feasibility of the system and analyzes the effects of various system parameters.

2 HYPERSPACE

2.1 Basis Selection

Creating a hyperspace that can provide fine-grain differentiation of different objects requires a careful selection of string labels as dimension bases (axes). In e-llama, the virtual coordinate of every object is the tuple created from its edit-distance to each of the axis labels. In essence, axis labels act as anchor points, and each component of an object's coordinate provides the distance of the object from that anchor point. Much like the Post Office metric on normed vector space [5], the distance from each anchor point clusters similar objects to the extent differentiable by that axis label, assigning them similar coordinates. However, a poor selection or an insufficient number of bases can assign similar coordinates to even dissimilar objects. For example, if axis labels are random strings, it is likely that each label will have a very similar edit-distance to any real English string of the same length. A careful selection of the axis labels is important, as the labels define the hyperspace in which keys will be clustered.

Labels that have some similarity to the actual objects in the system can help accentuate the differences in dissimilar objects. Intuitively, given sufficient labels, each object has a high probability of resembling some labels, the set of which is different and distinct from the set of labels resembling dissimilar objects. This intuition leads to simply selecting axis labels from actual objects. For example, in a deployment with movie titles as objects, a small random sampling of movie titles can serve as a set of axis labels. Once selected, axis labels do not need to change as long as the distribution of object names is relatively stable.

In addition to the selection of axis labels, the number of dimensions also plays an important role in creating an effective hyperspace for differentiating dissimilar objects. Increasing the number of bases should, intuitively, widen the separation of coordinates between dissimilar strings as it becomes increasingly unlikely for them to have the same edit-distance to a large number of independently chosen axis labels. The cost of additional bases is low, requiring only minimal increases in string storage and bandwidth. As we will see later in section 4, increasing the number of bases significantly improves the distinguishing power of the hyperspace yet incurs relatively little overhead.

Note that the Euclidean distance between the coordinates for two strings s_1 and s_2 is loosely related to the edit-distance between these two strings. In the worst case, these two strings might require $n = ||s_1|| = ||s_2||$ many insertions, deletions and replacements from the axis labels, and hence share the same coordinate, and the edit-distance between s_1 and s_2 might be $2n$. This bound forces proximity between related strings, and proper choice of independent axis labels forces dissimilar strings to acquire divergent coordinates in practice, as we show later in section 4.

2.2 Node ID Assignment

Similar to objects, nodes are assigned coordinates in hyperspace. Much like DHTs, each e-llama node is responsible for storing the set of objects for which it is the closest node. The assignment of node coordinates involves a subtlety. While any random assignment of coordinates to nodes will lead to a correct system that will work, the strong result that assures that routing will be performed in O($\log N$) hops [26] requires that nodes be distributed in hyperspace according to the same distribution as objects. To ensure that this is the case, node IDs in e-llama are determined by random sampling. Each node independently selects a random object name, determines its coordinate, and adopts that location as its identifier. Nodes ensure uniqueness by detecting coordinate collisions at join time. As in axis labels, node coordinates do not need to change as long as the distribution of object coordinates remains relatively stable.

3 ROUTING FRAMEWORK

The basic e-llama routing framework relies on multi-resolution rings to organize peers, a ring membership replacement scheme to maximize the usefulness of ring

members, and a gossip protocol for node discovery and membership dissemination. Finding the k data items with keys closest to the search key involves two phases. First, a multi-hop query routing protocol finds the closest node to the search key. Once the closest node is found, it recursively queries its nearby peers to determine the k closest data items.

3.1 Multi-Resolution Rings

The intuitive reason for the multi-resolution ring structure for organizing peers is to provide each node with near authoritative information on nodes that are near it, but also provide a sufficient number of out-pointers to far-away nodes to allow large hops and facilitate rapid search. Each e-llama node organizes its peers into a set of concentric rings centered on itself, where the ring distance is a measure of its Euclidean distance to the node. The ith ring has inner radius $r_i = \alpha s^{i-1}$ and outer radius $R_i = \alpha s^i$, for $i > 0$, where α is a constant, s is the multiplicative increase factor, and $r_0 = 0$, $R_0 = \alpha$ for the innermost ring. Each node keeps track of a finite number of rings; all rings $i > i^*$ for a system-wide constant i^* are collapsed into a single, outermost ring that spans the range $[\alpha s^{i^*}, \infty]$.

3.2 Ring Membership Management

The number of nodes per ring, p, represents a trade-off between accuracy and overhead. A large p allows each node to retain more information for better route selection during query routing, but requires additional overhead in both memory and bandwidth. The utility of a ring member is in relationship to the amount of diversity it can provide to the ring. For each ring, the node retains $p+l$ members, where l is a constant number of additional nodes that serve as potential ring candidates for use in the next ring membership selection process. During ring membership selection, an infrequent periodic event, the subset of p nodes from the $p + l$ members that forms a polytope with the largest hypervolume based on their coordinates are kept as ring members.

3.3 Gossip Based Node Discovery

A standard anti-entropy push protocol [10] provides node discovery and dissemination between e-llama nodes. At each gossip round, an e-llama node collects a random selection of its ring members, and sends this collection to a random member in each of its rings. The receiving node contacts each peer in the collection to discover their coordinates, and these peers are then stored as potential replacement for the node's current primary ring members.

3.4 Query Routing

Locating the closest node to the search key, the first phase in finding the k closest data items, involves a multi-hop search where each hop exponentially reduces the distance to the closest node. On receiving a query, an e-llama node determines its closest peer to the search key's coordinate. If the closest peer is closer to the key than the current node by some threshold, the node forwards the query to the peer. Otherwise, the node selects the closest peer or itself, whichever is closer to the key, as the closest node. This query routing protocol can find the closest node in $O(\log N)$ hops with high probability [25].

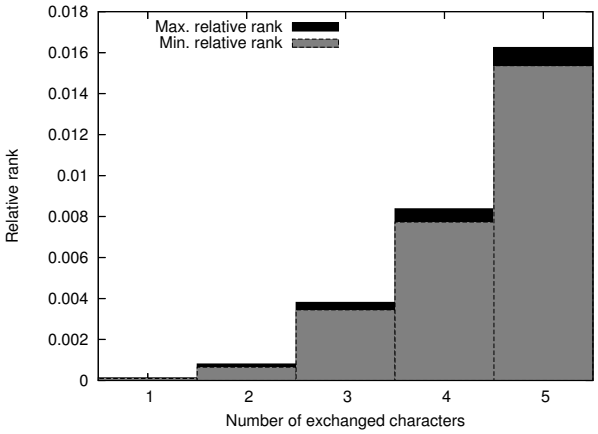


Figure 1: The relative rank of the actual string across different number of character exchanges. Less than 0.02 of the total entries need to be searched on average before finding the actual string for query strings with up to 5 characters exchanged.

3.5 K-Clustering

Once the closest node to the search key is found, e-llama determines all objects within a sphere centered around the key's coordinate, expanding the sphere until the k closest data items are inside. The protocol begins with the closest node asking all its neighbors within a distance of h from the search key's coordinate to recursively determine the k closest data items using the same query ID. Given that a node only responds to a search request once per query ID, the recursion terminates when all nodes within the sphere returns what each thinks is the k closest data items. The closest node receives up to k data items from each of its peers within the sphere, and determine the k actual closest items. The protocol repeats with a larger sphere if there are less than k data items within the previous sphere.

4 EVALUATION

We evaluate e-llama by applying it to synthetic searches based on the NetFlix database [3], which consists of a listing of 17770 movie titles. The first test consists of 1000 randomly chosen movie titles with a small number of characters exchanged to simulate typos and spelling variations. For the second test, 6552 randomly selected movie titles were modified with real human typos and misspellings from the Searchspell database [4].

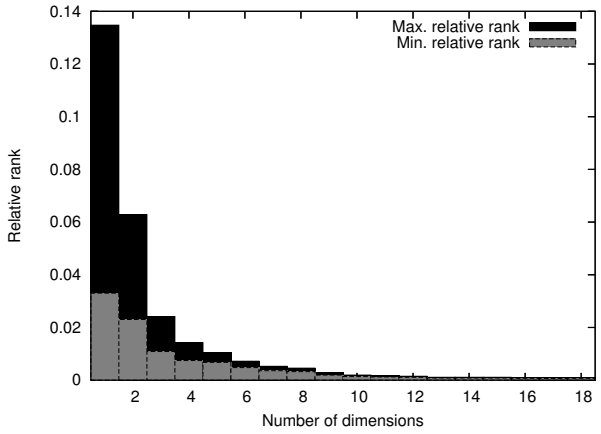


Figure 2: The relative rank of the actual string across different number of dimensions. A small increase in the number of dimensions can dramatically improve the relative rank.

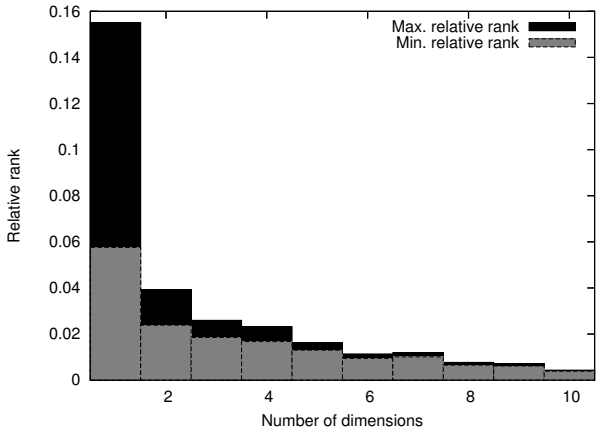


Figure 3: The relative rank of a movie title to one with human typos across different number of dimensions. As before, the relative rank decays rapidly as the number of dimensions increases.

We investigated several schemes for axis label selection which includes using random strings of both short and long length, using random English strings taken from a dictionary, and using randomly chosen titles from the movie database. We found that using movie titles as axis labels consistently outperforms the other two schemes, and use it exclusively for the rest of the evaluation.

We evaluate the effectiveness of e-llama's approximate match using the relative rank of the original string in the search results for the modified query string. We order the results of the approximate match based on the Euclidean distance of coordinates we defined earlier. Thus, if the original string has the lowest Euclidean distance to the query string, its absolute rank is 1; the absolute rank is 2 if one other string is closer to the query string, and so forth. The relative rank is the rank divided by the total number of movie titles in the database. Relative rank captures the average cluster size necessary to contain the desired object and thus the percentage of results that a user must check before locating the intended object. Since several titles commonly share the same distance to a modified query string, we show both the lowest and highest relative rank of the original string.

We first examine the change in relative rank from different numbers of perturbations to the original movie title. In this experiment, the number of dimensions is fixed at 20. Figure 1 shows that increasing the number of modifications significantly increases the difficulty of the problem. With five characters exchanged, the actual object has an average relative rank less than 0.02. In other words, the object resides within a cluster that contains less than two percent of the total number of movies on average. For perturbations of only one character, the average cluster size is less than 0.01 percent of the number of movie titles. These results suggest that the e-llama technique can return a very small cluster that nevertheless contains the desired object, even with very few dimensions in the hyperspace. A small cluster size limits the number of results the users must manually look through to find their actual object. A less encouraging result is the exponential growth in the relative rank from the linear increase in the number of perturbations.

Fortunately, Figure 2 shows that the relative rank also decreases exponentially with a linear increase in the number of dimensions. In this experiment, we fix the number of perturbations at two. With only one dimension, the relative rank is nearly 0.14. This suggests that the approximate string matching problem is not well suited to solutions using distributed hash tables which are restricted to a one dimensional space. However, with a modest increase from one to five dimensions, the relative rank falls significantly to approximately 0.01. The same phenomenon can be seen in Figure 3. The experiment consists of 6552 randomly selected movie titles, each of which has 80% of its words replaced by a mistyped or misspelled version from the Searchspell database [4] of human typos and spelling errors. On average, the edit distance between the queries and the actual titles is 4.9. In this setting, an increase from one to five dimensions decreases the relative rank to below 0.02. Additional dimensions require only small and inconsequential increases in bandwidth and computational overhead for most practical applications. Applications that receive search strings with a high number of perturbations can simply configure their deployment with a higher number of dimensions in order to arrive at their desired average relative rank.

5 RELATED WORK

E-llama is a distributed hash table that provides a novel approximate match primitive. It differs from previous DHTs [21, 23, 28, 19, 16, 12], which support only precise lookups.

The high dimensional hyperspace in e-llama is similar conceptually to virtual coordinate schemes for estimating inter-node latencies [17, 9, 14, 24, 22, 18]. However, increasing the number of dimensions entails very little associated cost in e-llama as the coordinates are completely synthetic and do not require network pings for assignment. The low cost of coordinate assignment and comparison also renders expensive techniques for reducing dimensions unnecessary.

Query routing in e-llama most closely resembles routing in Meridian [25], Small-World networks [13], and CAN [20]. In CAN, each node knows its immediate closest neighbor in each of the dimensions and greedily routes to the destination. However, border cases in dealing with churn makes CAN difficult to implement and deploy in practice. Small-World networks introduce long links between peers to reduce the number of routing hops to $O(\log^2 N)$. The query routing in e-llama is similar to Small-World network routing but reduces the number of hops to $O(\log N)$ by introducing additional structure. Meridian uses a similar multi-resolution ring structure as e-llama, but focuses on operating in non-grid like metric spaces and has no notion of absolute position of any of the nodes.

Efficient similarity comparison of strings are typically based on sampling techniques [11, 7, 15], where portions of a string represent the entire string. In these techniques, each string is broken down into a number of overlapping sub-strings of fixed length and each individual sub-string is hashed. A consistent sampling of approximately k sub-string hashes is taken from each string as fingerprints, and the *resemblance* of two strings is the number of shared sub-strings in the fingerprint, divided by the total number of unique sub-strings from the fingerprints. These schemes differ from e-llama as they require a centralize system for performing string comparison, where e-llama provides a distributed and peer-to-peer solution.

In [27], the authors use the Soundex algorithm to encode keywords by their phonemes before indexing them in a DHT. Unlike edit distance, Soundex is appropriate only for English keywords and is not effective against typing errors.

6 SUMMARY

In this paper, we described a new technique for efficient approximate matching in peer-to-peer overlays. The technique is scalable, efficient, and of immediate applicability to domains, such as peer-to-peer filesharing, where query terms are provided by users and require an approximate match against objects in the system. More generally, we presented an object clustering technique based on creating a synthetic, high-dimensional space and assigning coordinates to objects in this space where Euclidean distances capture similarity. Given appropriately chosen axis labels, such a mapping can facilitate the identification of similar objects. We showed how coupling such a space with an efficient search function based on small-worlds can yield an efficient and scalable system. This overall approach may be applicable to other domains where a similarity-based clustering of objects is desired.

References

[1] Gnutella. http://www.gnutella.com/.

[2] Britney Spears spelling correction. http://www.google.com/jobs/britney.html.

[3] Netflix Prize. http://www.netflixprize.com.

[4] Searchspell. http://www.searchspell.com/typo/.

[5] Metric space. http://en.wikipedia.org/wiki/Metric_space.

[6] A. Bharambe, M. Agrawal and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *SIGCOMM*, Portland, Oregon, August 2004.

[7] A. Broder, S. Glassman and M. Manasse. Syntactic Clustering of the Web. In *World Wide Web Conference*, Santa Clara, California, April 1997.

[8] A. Crainiceanu, P. Linga, J. Gehrke and J. Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. In *WebDB*, Paris, France, June 2004.

[9] F. Dabek, R. Cox, F. Kaashoek and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM*, Portland, Oregon, August 2004.

[10] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, Vancouver, Canada, August 1987.

[11] N. Heintze. Scalable Document Fingerprinting. In *Workshop on Electronic Commerce*, Oakland, California, November 1996.

[12] F. Kaashoek and D. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In *IPTPS Workshop*, Berkeley, California, February 2003.

[13] J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *STOC*, Portland, Oregon, May 2000.

[14] H. Lim, J. Hou and C. Choi. Constructing Internet Coordinate System Based on Delay Measurement. In *IMC*, Miami Beach, Florida, October 2003.

[15] U. Manber. Finding Similar Files in a Large File System. In *Winter Technical Conference*, San Francisco, California, January 1994.

[16] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS Workshop*, Cambridge, Massachusetts, March 2002.

[17] T. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *INFOCOM*, New York, New York, June 2002.

[18] M. Pias, J. Crowcroft, S. Wilbur, T. Harris and S. Bhatti. Lighthouses for Scalable Distributed Location. In *Intl. Workshop on P2P Systems*, Berkeley, California, February 2003.

[19] S. Ratnasamy, P. Francis, M. Hadley, R. Karp and S. Shenker. A Scalable Content-Addressable Network. In *SIGCOMM*, San Diego, California, August 2001.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. A Scalable Content-Addressable Network. In *SIGCOMM*, San Diego, California, August 2001.

[21] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, Heidelberg, Germany, November 2001.

[22] Y. Shavitt and T. Tankel. Big-Bang Simulation for Embedding Network Distances in Euclidean Space. In *INFOCOM*, San Francisco, California, March 2003.

[23] I. Stoica, R. Morris, D. Karger, F. Kaashoek and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*, San Diego, California, August 2001.

[24] L. Tang and M. Crovella. Virtual Landmarks for the Internet. In *IMC*, Miami Beach, Florida, October 2003.

[25] B. Wong, A. Slivkins and E. G. Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *SIGCOMM*, Philadelphia, Pennsylvania, September 2005.

[26] B. Wong, A. Slivkins and E. G. Sirer. A Framework for Network Location-Aware Node Selection. *TOCS (in submission)*.

[27] M. Zaharia, A. Chandel, S. Saroiu and S. Keshav. Finding Content in File-Sharing Networks When You Can't Even Spell. In *Intl. Workshop on P2P Systems*, Bellevue, Washington, February 2007.

[28] B. Zhao, J. Kubiatowicz and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Berkeley, California, April 2001.

Optimizing Power Consumption in Large Scale Storage Systems

Lakshmi Ganesh, Hakim Weatherspoon, Mahesh Balakrishnan, Ken Birman
Computer Science Department, Cornell University
{lakshmi, hweather, mahesh, ken}@cs.cornell.edu

Abstract

Data centers are the backend for a large number of services that we take for granted today. A significant fraction of the total cost of ownership of these large-scale storage systems is the cost of keeping hundreds of thousands of disks spinning. We present a simple idea that allows the storage system to turn off a large fraction of its disks, without incurring unacceptable performance penalties. Of particular appeal is the fact that our solution is not application-specific, and offers power-savings for a very generic data center model. In this paper, we describe our solution, identify the parameters that determine its cost-benefit tradeoffs, and present a simulator that allows us to explore this parameter space. We also present some initial simulation results that add weight to our claim that our solution represents a new power-saving opportunity for large-scale storage systems.

1 Introduction

The declining costs of commodity disk drives has made online data storage a way of life. So much so that companies like Google and Yahoo host hundreds of thousands of servers for storage. However, there is a catch: a hundred thousand servers consume a lot of power! Not only does this translate to many millions of dollars annually on electricity bills, the heat produced by so much computing power can be searing. An article in The New York Times describes one of Google's data centers: ".. a computing center as big as two football fields, with twin cooling plants protruding four stories into the sky"[9]. Conclusion: Power conservation is an important concern for big server clusters. Since disks account for a significant fraction of the energy consumed[6], several approaches for disk power management have been proposed and studied. We will examine some of these here. But first let us lay out some of the groundwork.

Any disk power management scheme essentially attempts to exploit one fact: disks can be run in high-power mode, or low-power mode, with a corresponding performance tradeoff. In the limit, a disk can be shut off so that it consumes no power. Given a large cluster of disks, only a fraction of them is accessed at any time, so that the rest could potentially be switched to a low-power mode. However, since mode transitions consume time and power, disk management schemes have to walk the tightrope of finding the right balance between power consumption and performance.

The solution space explored thus far in the literature can be divided as follows: (1) Hardware-based solutions, (2) Disk Management solutions, and (3) Caching solutions. Each of these solutions proposes a new system of some kind; *hardware-based solutions* propose novel storage hierarchies to strike the right balance between performance and power consumption; *disk management solutions* interject a new 'disk management layer' on top of the file system, which controls disk configuration and data layout to achieve power-optimal disk access patterns; *caching solutions* devise new power-aware caching algorithms that allow large fractions of the storage system to remain idle for longer periods of time, allowing them to be switched to lower power modes.

The principal contribution of this paper is to argue that there is a fourth niche as yet unexplored: (4) File System solutions. We do not present a new system; instead, we take an idea that has been around for well over a decade now - the Log-Structured File System (LFS) [13] and argue that technological evolution has given it a new relevance today as a natural power-saving opportunity for large-scale storage systems. The key insight is that, where other solutions attempt to predict disk access to determine which disks to power down, the LFS automatically provides a perfect prediction mechanism, simply by virtue of the fact that all write-accesses go to

the log head. Section 3 explains and expands on this idea.

1.1 Idea Overview

To see why LFS is a natural solution to the problem of disk power management, consider some of the challenges involved:

- Short Idle Periods: Server systems typically are not idle long enough to make it worthwhile to incur the time+power expense of switching the disk to a low-power mode, and switching it back when it is accessed. This is a notable point of difference between server systems and typical mobile device scenarios (like laptops), which makes it hard to translate the solutions devised for mobile devices to server systems. As we shall see, LFS localizes write-access to a small subset of disks; this feature, when combined with a cache that absorbs read-accesses, results in long disk idle periods.

- Low Predictability of Idle Periods: Previous studies [7] have shown that there exists low correlation between a given idle period's duration and the duration of previous idle periods. This variability makes it difficult to devise effective predictive mechanisms for disk idle times. The LFS neatly circumvents this problem by predetermining which disk is written to at all times.

- Performance Constraints: Server systems are often constrained by Service Level Agreements to guarantee a certain level of performance, so that finding a solution that provides acceptable performance to only a fraction of the incoming requests (albeit a large fraction) may often not be sufficient. As we shall show, the LFS provides an application-independent solution that allows the system to perform consistently across a wide range of datasets.

- The law of large numbers: Large scale server systems process incredibly large request loads. Directing these to a small fraction of the total number of disks (the fraction that is in 'high-power mode') can significantly raise the probability of error and failure. The fact that the disks used in these contexts are typically low-end with relatively weak reliability guarantees, exacerbates this problem. As we shall see, our solution alleviates this problem by making sure that the live subset of disks is not constant.

The rest of this paper is organized as follows: Section 2 describes some of the solutions explored in the first three quadrants mentioned above. Section 3 presents and analyzes our solution, while Section 4 discusses our evaluation methodology and results. We conclude in Section 5.

2 Related Work

Hardware-based Solutions
The concept of a memory hierarchy arose as a result of the natural tradeoff between memory speed and memory cost. Carrera et. al. point out in [1] that there exists a similar tradeoff between performance and power-consumption among high-performance disks and low-performance disks such as laptop disks. They explore the possibility of setting up a disk hierarchy by using high- and low-performance disks in conjunction with each other. In a related vein, Gurumurthi et. al.[8] propose Dynamic Rotations Per Minute (DRPM) technology, whereby disks can be run at multiple speeds depending on whether power or performance takes precedence. DRPM, however, poses a significant engineering challenge whose feasibility is far from obvious.

Another approach is proposed by Colarelli et. al. in [2], using massive arrays of inexpensive disks (MAID). They propose the use of a small number of *cache* disks in addition to the MAID disks. The data in these cache disks is updated to reflect the workload that is currently being accessed. The MAID disks can then be powered down, and need only be spun up when a cache miss occurs, upon which their contents are copied onto the cache disks. This approach has several of the weaknesses that memory caches suffer, only on a larger scale. If the cache disks are insufficient to store the entire working set of the current workload, then 'thrashing' results, with considerable latency penalties. Further, the cache disks represent a significant added cost in themselves.

Disk Management Solutions
Pinheiro and Bianchini [11] suggest that if data is laid out on disks according to frequency of access, with the most popular files being located in one set of disks, and the least popular ones in another, then the latter set of disks could be powered down to conserve energy. Their scheme is called Popular Data Concentration (PDC) and they implement and evaluate a prototype file server called Nomad FS, which runs on top of the file system and monitors data layout on disks. Their findings are that if the low-access disks are powered down, this results in a considerable performance hit; they suggest instead that they be run at low speed. While their idea is sound, it is not clear whether this scheme would adapt to different workloads.

Son et. al. propose another data layout management scheme to optimize disk access patterns [14]. Their approach uses finer-grained control over data layout on disk, tuning it on a per-application basis. Applications are instrumented and then profiled to obtain array access sequences, which their system then uses to determine optimal disk layouts by computing optimal stripe factor, stripe size, start disk etc. Again, the wisdom of marrying the disk layout to the application seems questionable.

Hibernator, proposed by Zhu et. al [6], combines a number of ideas. It assumes multispeed disks, and computes online the optimal speed that each disk should run at. To minimize speed transition overheads, disks maintain their speeds for a fixed (long) period of time - they call this the coarse-grained approach. Hibernator includes a file server that sits on top of the file system and manipulates data layout to put the most-accessed data on the highest speed disks. The authors address the issue of performance guarantees by stipulating that if performance drops below some threshold, then all disks are spun up to their highest speed.

Caching Solutions
Zhu et. al [5] observe that the storage cache management policy is pivotal in determining the sequence of requests that access disks. Hence, cache management policies could be tailored to change the average idle time between disk requests, thus providing more opportunities for reducing disk energy consumption. Further, cache policies that are aware of the underlying disk management schemes (eg. which disks are running at which speeds, say) can make more intelligent replacement decisions. The authors present both offline and online power-aware cache replacement algorithms to optimize read accesses. They also show through experiments the somewhat obvious fact that for write accesses, write-back policies offer more opportunities to save power than write-through policies.

3 A New Solution

We shall now argue that there remains an unexplored quadrant in this solution space. Caches are used to minimize accesses to disk. Good caching algorithms practically eliminate read accesses to disk. However, write accesses (whether synchronous or not) must still eventually access the disk. Thus, assuming perfect caching, disk access will be write-bound. Putting a disk management layer on top of the file-system to optimize data layout for writes is only halfway to the solution. To take this idea to its logical conclusion, it is necessary to rethink the file system itself. In the context of write-access optimization, a very natural candidate is the log-structured file system [13]. We now give a brief overview of the log-structured file system before describing the power-saving opportunity it represents.

3.1 Log-Structured File System

The Log-Structured File System (LFS) was motivated by a need to optimize the latency of write-accesses. Writing a block of data to a Seagate Barracuda disk costs about 11.5ms in seek time and 0.025ms/KB in transmission time. The key observation here is that seek time is a large and *constant* term in latency computation. To eliminate this term, the LFS replaces write operations by append operations. Secondary storage is treated as a large append-only log and writes always go to the log head. Seek time is thus eliminated, and write latency becomes purely a function of the disk bandwidth.

How do reads in the LFS work? In the same way as in conventional file systems! Reads require random-access, and hence do not avoid seek-latency. However, the assumption is that with good caching mechanisms, reads will be a small fraction of disk accesses.

As can be imagined, space reclamation is a tricky problem in log structured file systems. However, excellent solutions have been proposed to solve it, and one such is of interest to us: the disk is divided into large log segments. Once a log segment gets filled, a new log segment is allocated and the log head moves to the new segment. When some threshold of a segment gets invalidated, its valid data is moved to another segment (replacing that segment's invalid data), and it is then added to the pool of free log segments. Over time, this process results in a natural division of allocated segments into stable (ie.. consisting almost entirely of data that is rarely invalidated/modified), and volatile ones (which need to be constantly 'cleaned'). We will see how this feature can be used to save power.

3.2 LFS: A Power-Saving Opportunity

The disk-management policies described in the related works section essentially attack the problem by trying to predict in advance which disk any given access will go to. They optimize the data layout on disks to ensure that accesses are localized to some fraction of the disks, so that only these need be powered up. However, these are all probabilistic models - a new access has some probability of not fitting this model and needing to access a powered-down disk. Further, in such schemes, disk layout becomes tied to particular applications; two applications that have completely different access

patterns might require different data layouts on disk leading to conflicts that reduce possible power-savings.

Since all writes in an LFS are to the log head, we know in advance which disk they will access. This gives us the perfect prediction mechanism, at least for write-accesses. Besides being deterministic, this prediction mechanism is also application-independent. Thus, if most accesses to disks were writes, we could power down every disk but the one that the log head resides on. This, however, is an ideal case scenario. Our view is that, with a good caching algorithm (the power-aware caching algorithms described in the related works section are good candidates), reads to disk can be minimized, and only a small fraction of the disks need be powered on in order to serve all writes as well as reads.

However, what about the performance and power costs of log cleaning? Matthews et. al present some optimizations in [4] to hide the performance penalty of log cleaning even when the workload allows little idle time. The power costs of log cleaning are a little more tricky to justify; however, this is where the natural division of segments into stable and volatile ones that the log cleaning process results in (as described above) can help. After a significant fraction of segments on a disk have been classified as stable, volatile, or free, we power the disk on and copy the stable segments to a 'stable' disk, volatile segments to a 'volatile' disk (disk is kept on), and the entire disk is freed for reuse. This is similar to the log cleaning scheme described in [10], which uses a 'hidden' structure embedded in the log to track segment utilization. Cleaning an entire disk amortizes the cost of powering it on.

4 Evaluation

4.1 Methodology

We have proposed the use of LFS in lieu of conventional file systems in data-center scenarios to achieve power conservation. For this idea to be accepted, two questions need to be answered in the affirmative: (1) *Does this new scheme result in significant power savings?*, and (2) *Does this new scheme provide comparable performance to existing schemes?* Further, the answers to these questions must be largely application-independent, and must apply to a generic data center model. To address these questions, we present a simulator - Logsim. Logsim consists of less than a thousand lines of Java code and is a single-threaded, discrete event-based simulator of a log-structured file system. Given a trace of read and write requests, Logsim returns the observed access latencies, disk utilization, cache-hit ratio, disk-mode transitions etc., for the chosen set of configuration parameters. We use real-world traces for our simulations from a web-server that serves images from a database[12]. Table 1 describes the characteristics of a sample trace. While a true evaluation of the feasibility and efficacy of our solution can only be achieved through an actual implementation, simulation provides an elegant way to identify and explore some of the cost-benefit tradeoffs in a scaled-down version of our system.

Number of accesses	476884
Number of files touched	23125
Number of bytes touched	4.22GB
Average number of bytes/access	8.8 KB

Table 1: Sample Trace Characteristics.

The mechanism we simulate is as follows: All (non-empty) disks are assumed to begin in the 'on' state, and an access count (an exponentiated average) is maintained for each disk. The user specifies the maximum percentage (m) of disks that are kept powered on. Periodically (200ms, in our experiments), a 'disk check' process scans the access count for each disk and powers down all but the most-accessed top $m\%$ of the disks, as well as any disk which does not have at least t access count. t is 0 in our experiments. If a cache-miss results in an access to a powered-down disk, then this disk is spun up (to remain powered on until the next 'disk check'), and there is a corresponding latency penalty. Judicious choice of m and t minimizes the probability of this occurrence.

4.2 Results

To save power, we must turn off some percentage of disks in the storage system. However, there are two opposing forces at play here. A large number of powered-on disks results in good performance (low latency), but also low power savings. On the other hand, decreasing the number of powered-on disks incurs two possible penalties: increased latencies, and increased mode-transitions. Mode-transitions consume power and thus counter the potential savings achieved by powered-down disks. To find the optimal percentage of disks to be powered down, we ran a set of simulations on Logsim and varied the number of disks that we kept powered up from none (except the log-head disk), to all, in steps of 20%. Thus, out of a total of 66 disks, 1, 13, 26, 39, 52, and 66 disks were kept powered up, respectively. For each run, we examine both its performance (in terms of observed access latencies), as well as its power-consumption. Fig. 1, 2 and 3 show the results of these simulations.

The performance of our system depends heavily on its cache configuration. Since cache optimization is

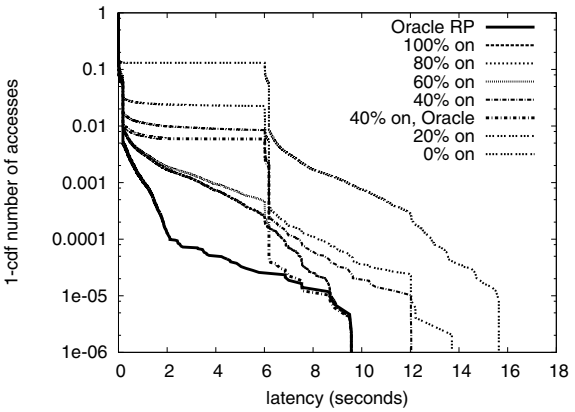


Figure 1: CCDF: Effect of increasing percentage of powered-up disks on performance.

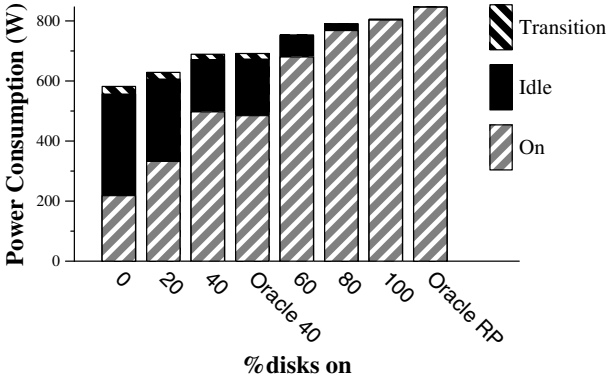


Figure 2: Effect of increasing percentage of powered-up disks on power consumption

an orthogonal issue that comprises an entire field of research in itself, it is important to isolate its effect on performance. To achieve this, we implemented an 'ideal cache' algorithm, which we term the *oracle*. Experiments using the oracle approximate the best performance we could achieve since an oracle has future knowledge and is able to replace items accessed furthest in the future [3]. In fig. 1, 2, 3, the data points that use this algorithm are annotated with the word 'Oracle'.

Finally, we also wish to compare our system against conventional (not log-structured) file systems. As an approximation of such a system, we implemented a 'random placement' (RP) algorithm, which maps each block to a random disk. All disks are kept powered up, and ideal caching (oracle) is assumed. This data point is labeled 'Oracle RP' in our graphs.

Having set the context, let us examine our results. Fig. 1 shows the relation between performance (per-access latency) and the number of disks that are powered on. If we imagine a line at y=.001 (ie.. 99.9% of the accesses live above this line) 60% disks ON is the third best configuration, next only to the Oracle RP and 100% disks ON configurations. Further, the performance degradation in going from 100% disks ON to 60% disks ON is negligibly small. The principal take-away is that, for the system under test, the optimal configuration is to have 60% of the disks powered on. In other words, 40% of the disks can be spun down while still maintaining performance comparable to that of a conventional file system.

Fig. 2 shows an estimate of the actual power savings achieved by our solution. The height of each bar is the average power consumed while processing the trace. Further, each bar shows the break-up of power consumed by powered-up disks (On), powered-down disks (Idle), and mode-transitions (Transition). We assume the following disk specifications: Avg. operating power = 12.8 W, Avg. idle power = 7.2 W, Avg. mode transition power = 13.2 W, Avg time for transition = 6s. We see that turning off 40% of the disks results in 12% power savings (as compared to 32% with all the disks off), while maintaining acceptable performance.

Finally, fig. 3 shows how much time the disks spend in on/off/transition states. The height of each bar is the cumulative time spent by each disk in each of these three states. When 0% disks are on, the run takes 7253 cumulative hours; we omit this bar from our graph for clearer presentation. We see that, both the total duration of the experiment, as well as the number of mode-transitions, increase as the percentage of disks that is powered on is decreased. However, as in fig. 1, we see that keeping 60% disks on strikes an acceptable balance.

5 Conclusion

In this paper, we point out a new opportunity for saving power in large-scale storage systems. The idea is elegant in its simplicity: log structured file systems write only to the log head; as a result, if read accesses are served by the cache, then write accesses touch only the log head disk, potentially allowing us to power down all the other disks. Existing solutions like disk management solutions and caching solutions are typically application-specific; our solution, on the other hand, is applicable to any cacheable dataset. Since existing solutions are typically layered on top of the file-system, they could be used in conjunction with our solution to take advantage of application-specific optimizations.

We also provide some initial simulation results that

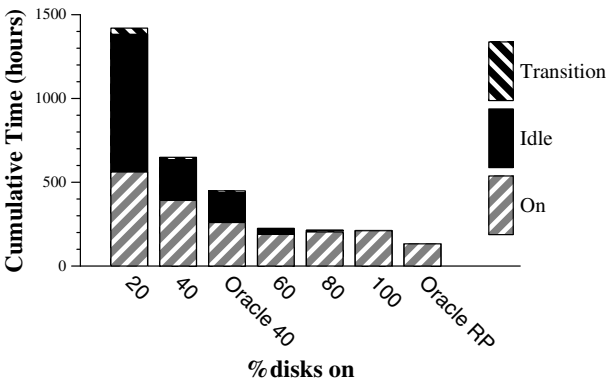


Figure 3: Effect of increasing percentage of powered-up disks on trace running time.

validate our claim that power-savings are possible using a log-structured file system. While simulations cannot provide conclusive evidence for the feasibility of a system, they are an effective means to identify promising solutions. Our principal contribution in this paper is in having shown a new fit for an old idea; we believe that the log-structured file system shows promise as a power-saving opportunity for large-scale storage systems.

In future work, several questions still remain to be addressed. Our evaluation has been limited by the difficulty of obtaining real filesystem traces from commercial data centers; we are actively looking for more recent traces to test our solution against. We are also working on a more thorough study of the efficacy of the log cleaning approach we outline here. Finally, we believe the LFS provides an interesting substratum to build more elaborate solutions on, and we are working on some promising options that we hope to share with the community soon.

Acknowledgments

This work was partially funded by Intel Corporation and the National Science Foundation. Special thanks to Saikat Guha for his input in the simulator design. We also wish to thank our anonymous reviewers for their valuable feedback.

References

[1] E. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *ICS '03: Proceedings of the 17th International Conference on Supercomputing*, June 2003.

[2] D. Colarelli, D. Grunwald, and M. Neufeld. The Case for Massive Arrays of Idle Disks (MAID). In *The 2002 Conference on File and Storage Technologies*, 2002.

[3] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, 1968.

[4] J. Matthews et. al. Improving the performance of log-structured file systems with adaptive methods. In *SOSP '97*, October 1997.

[5] Q. Zhu et. al. Reducing energy consumption of disk storage using power-aware cache management. *HPCA*, 00:118, 2004.

[6] Q. Zhu et. al. Hibernator: helping disk arrays sleep through the winter. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[7] S. Gurumurthi et. al. Interplay of energy and performance for disk arrays running transaction processing workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2003.

[8] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. Reducing disk power consumption in servers with drpm. In *IEEE Computer*, Dec 2003.

[9] J. Markoff and S. Hansell. Hiding in Plain Sight, Google Seeks More Power. In *The New York Times, Online Ed., June 14*, 2006.

[10] Oracle. Berkeley db java edition architecture. An Oracle White Paper, September 2006. http://www.oracle.com/database/berkeley-db/je/index.html.

[11] Eduardo Pinheiro and Ricardo Bianchini. Energy conservation techniques for disk array-based servers. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, 2004.

[12] D. Roselli and T. Anderson. Characteristics of file system workloads. In *UCB/CSD Dissertation*, 1998.

[13] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[14] S. W. Son, G. Chen, and M. Kandemir. Disk layout optimization for reducing energy consumption. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, 2005.

Can Ferris Bueller Still Have His Day Off? Protecting Privacy in the Wireless Era

Ben Greenstein* Ramakrishna Gummadi† Jeffrey Pang§
Mike Y. Chen* Tadayoshi Kohno‡ Srinivasan Seshan§ David Wetherall‡*
*Intel Research Seattle †University of Southern California
‡University of Washington §Carnegie Mellon University

ABSTRACT

Today's rich and varied wireless environment, including mobile phones, Wi-Fi-enabled laptops, and Bluetooth headsets, poses threats to our privacy that cannot be addressed with existing protocols. By considering 802.11 as a case study and analyzing publicly available 802.11 traces, we show that a device can be identified and tracked over time through its persistent link-layer address, list of known networks (SSIDs), and other protocol and physical layer characteristics. We argue that it is in the best interest of providers as well as users to design systems that maintain user privacy. We identify several research challenges to doing so and offer some direction towards a solution.

1 INTRODUCTION

Many people now have several wireless devices, and new ones—like the Wii controller and the Nike+iPod pedometers—are regularly being introduced and adopted. Perhaps the biggest reasons for their success are that they support mobility and are convenient. With more and more locations providing wireless services (*e.g.,* metropolitan-scale 802.11 networks are deployed in Taipei and are under way in San Francisco, Moscow, and many other cities [21]), users can be mobile while staying connected.

This heightened level of wireless connectivity brings many advantages, yet it also threatens our privacy in new ways that are underappreciated. It is well-known that wireless links are more exposed than their wired counterparts, as messages are broadcast to anyone within radio range (250 meters on 802.11b with standard antennas, and much farther to receivers with directional ones [19]). To counter the threat of eavesdropping and provide confidentiality comparable to what one might expect on a wired network, privacy mechanisms such as WEP/WPA for 802.11 and A5/1 for GSM encrypt packet contents. Although such mechanisms have been plagued with design and implementation flaws [6, 7], we take the optimistic view that the problem of how to build secure links is solved from the point of view of research and we do not consider it further in this paper.

The new threats of wireless devices are highlighted by recent concerns about *location privacy*. This risk stems from the mobility of wireless devices. The fear is that people can be tracked every second of every day, and with high accuracy, via the devices they carry. Location-aware systems, such as Active Badge [25] and RADAR [3], which explicitly estimate and communicate device positions, have spurred significant research in the mobile and ubiquitous computing communities focused on preventing the unauthorized disclosure of this information. Solutions include using centralized access controls (*e.g.,* [13, 14, 22]), having sensors perturb collected data before it is stored [12], and enabling users to discover their own locations using only passive measurements [18].

Location privacy is also threatened by systems that do not provide location awareness explicitly; any wireless device may betray who and where a user is, because transmissions often contain unique identifiers (*e.g.,* RFID tags [16]) or addresses (*e.g.,* 802.11 and Bluetooth devices [11, 15, 28]) that can be observed by anyone nearby. Moreover, masking these identifiers using *pseudonyms* [11, 15] or *mix zones* [5] is problematic in practice when they are also used for authentication and/or billing, as MAC addresses are in some 802.11 networks.

We argue that location privacy threats should all be viewed as facets of a larger wireless privacy problem; the RFID, 802.11 and Bluetooth tracking threats are essentially equivalent in their reliance on eavesdropping to discover unique identifiers or addresses. This threat should be recognized whether locations are made explicit as part of the operation of the system or can be unintentionally inferred. And it is posed (to different degrees) by service providers and legitimate users as well as third-party eavesdroppers. Note that current best security practices for data confidentiality do not alleviate these threats.

Moreover, wireless privacy threats are broader than the tracking of specific individuals. This is because even more limited information leaks can be of concern. As wireless devices become more diverse and more specialized in their individual function, it becomes easier to infer what kind of devices or applications are in use at a particular location, irrespective of the identity of the user. This threat (known as inventorying in the context of RFID) may be used to profile people, *e.g.,* for health via wireless medical or fitness devices, or target individuals, *e.g.,* for theft of Zunes or expensive home entertain-

ment systems. Even the detection of many forms of wireless communication leaks valuable information about the nearby environment due to small transmission ranges. Monitoring may reveal whether it is likely that people are present; this may violate users' natural assumptions about the privacy of their environments, *e.g.,* when their presence in their homes is detected from outside. Monitoring also may be used to estimate population or activity levels [23].

We argue that any systematic treatment of wireless privacy must encompass all of the above threats rather than view them piecemeal. Efforts to do so will face several challenges. They must articulate the problem in a way that goes beyond isolated examples of privacy failures; we hope this position paper contributes to this step. Then they must devise solutions that protect privacy and cause them to be put into practice. We make two points relevant to these steps. The first is that technical solutions are likely to play a large role in solutions in addition to legal, regulatory and social mechanisms. Wireless privacy threats are a technical creation, and we use a case study of 802.11 in this paper to argue that technical changes in the design of wireless protocols can offer some immediate relief. This is beneficial because it sidesteps the costs of other solutions, and because legal deterrents are less likely to be viable for many forms of wireless communications, *e.g.,* in the unlicensed ISM band versus cellular systems.

The second point is that, while privacy is often portrayed as a user concern that is unevenly appreciated, it is in the interests of providers to deploy systems that respect privacy. This is because there are legal and financial risks associated with the disclosure of confidential customer or employee information, whether by accident, theft, or subpoena. This is apparent from the public outcry surrounding privacy invasions, *e.g.,* the AOL release of Web search terms that identified users and the Benetton RFID boycott, and regulatory requirements on some businesses such as telecommunications providers. Moreover, companies are increasingly developing policies to protect the private information of their employees in all facets of operations as society is increasingly aware of privacy threats.

In the remainder of this paper, we highlight possible wireless privacy threats using an analysis of publicly available 802.11 traces, describe measures that can begin to improve privacy, and discuss some of the longer-term technical challenges that must be addressed. We choose 802.11 for our case study because our results, and in particular a new privacy attack that we uncover, suggest that the privacy problem for wireless networks may be more complex than originally anticipated.

2 802.11 Case Study

To make our discussion concrete, we sketch several scenarios that highlight existing privacy threats in the context of 802.11 networks. Each scenario illustrates a different type of threat and exposes a different privacy leakage vector. These fictional scenarios involve FooNet, a metropolitan-scale network provided by Foo, and two employees of major companies: Ferris from Foo and Boris from Bar. The threats range from individuals (both users of FooNet and other parties in the vicinity) to providers (Foo and FooNet in this case) as both victim and attacker.

To back up these scenarios, we studied 802.11 traces taken at SIGCOMM in 2004 to understand what information is leaked. These traces have been anonymized to protect the identities of the attendees: client MAC addresses were consistently hashed and the contents of data frames were removed. In our exercise, we let the hashed MAC addresses identify individual clients, and assume that the payload is not available to any parties other than the client and access point (AP). The latter would be the case if encryption keys were established per client as in WPA2, but admittedly this was not used at SIGCOMM 2004.

Scenario 1: Provider threat to individual privacy. Ferris bought a ZuNod, a Wi-Fi-enabled portable music player, during lunch on Monday, and spent the afternoon at work setting it up. By the end of the day he was able to connect the ZuNod to Foo's corporate Wi-Fi network, authenticate using his corporate username and password, and download songs from his favorite site.[1] On Tuesday morning, Ferris calls in sick to spend the day listening to music while strolling around the city. He subscribes to FooNet's free Internet service to download songs while mobile.

For amusement, an overeager Foo human resources associate decides to search for MAC addresses that connect to both Foo's corporate network and FooNet. The result isn't particularly interesting. A lot of employees use FooNet. However, when he restricts the results to the MAC addresses used by employees who have called in sick, he finds Ferris. Moreover, on closer inspection, he sees that Ferris spent much of the day in the city park and at the art museum. Nothing is done with this information as it is clearly an invasion of Ferris' privacy. However, by coincidence, Ferris is fired two weeks later. Ferris' lawyers subpoena HR records, find the MAC address search, sue Foo for invasion of privacy, and a media storm ensues.

Problem 1: Persistent addressing. The above scenario is possible because 802.11 interfaces broadcast persistent and globally unique MAC addresses, which is a known privacy problem. This persistence makes them easy to use to deliver packets, and their uniqueness helps avoid problems associated with naming collisions. However, such addresses also allow separate observations of the same client at different places and times to be tied to-

[1] The ZuNod is a hypothetical Zune-like device with unfettered Wi-Fi access. The initial Zune restricts Wi-Fi usage to local music sharing.

gether. This allows not only tracking by any observer [15] but linkage of other databases. This is what enables Ferris to be profiled even though he maintains corporate and FooNet accounts that are otherwise unrelated. Clearly, if we have any hope of providing wireless privacy, we must obscure these addresses.

Scenario 2: Individual threat to individual privacy. Boris' job in the venture capital group at Bar is to meet with startups and decide which ones to fund. He'll usually arrange several onsite visits to those startups that have compelling business plans. When away from his office, Boris opportunistically connects to available open networks and uses a VPN to encrypt his data.

Before being fired, Ferris had the same job at Foo and had an uncanny knack for beating Boris to the good startup opportunities. Ferris and Boris often ate lunch at a restaurant that provides free Wi-Fi access. While Boris would spend his lunches catching up on e-mail and reading the news, Ferris would spend them monitoring to see to which networks Boris's computer tries to connect. This would tip Ferris off to likely startup locations, which he would later visit to look for funding candidates.

Problem 2: Exposed resource discovery. We discovered that the above scenario is possible through trace analysis. Many 802.11 clients actively scan for specific networks to which they have connected in the past. They do this by sending probe request frames, each of which contains the SSID name of one of the networks that it prefers to use. These SSIDs are sent in the clear. This probing behavior is the default for Windows XP as it speeds up network discovery and provides a way to associate with networks that don't periodically announce their existence. Hence Ferris can observe Boris' preferred SSIDs by listening to his transmissions; any observer can do this despite data encryption. Ferris can then obtain the likely street addresses by looking the SSIDs up in a Wi-Fi location database such as Wigle [27].

This scenario demonstrates that while hiding addresses is necessary to provide wireless privacy, it isn't sufficient because other kinds of names are exposed. For active scanning in 802.11, the network names are often revealing, even without a database such as Wigle, since providers often choose meaningful names. Thus when a user probes for the networks that he has connected to previously, in effect, he advertises where he *has been*; *i.e.,* an attacker could use this technique to compromise a victim's *past* privacy.[2] Moreover, an unusual SSID or set of SSIDs can alert an observer to the presence of a particular user, regardless of whether the user's MAC address is changed frequently. Active scanning was previously known to be a security flaw as it can facilitate hijackings [20], but it had not been associated with privacy leaks as best we know. In the traces we examined, 161 users emitted a network name that was unique to a single user. 460 of the 566 users in the trace advertised the names of their preferred networks.

[2] We actually found this vulnerability when we noticed that the SSID of one of our own home networks was evident in the trace! Looking further, we were surprised to find that the trace named other networks at universities and companies that one of us had visited before attending SIGCOMM 2004.

Scenario 3: Provider/individual threat to provider privacy. Bar plans to compete with FooNet by launching its own metropolitan scale network. In an effort to take advantage of Foo's successes and mistakes, Bar decides to analyze FooNet to learn how many users connect to it, how much traffic they generate, and so forth. It is in Foo's best interest to keep these statistics to itself to prevent Bar from gaining a second mover advantage. To monitor FooNet, Bar deploys a much smaller number of mobile nodes that drive by FooNet hotspots at various times of day. Similarly, Boris might drive around himself and provide this information to one of his startup ventures. A comparable threat may be looming in the city-wide 802.11 joint venture in San Francisco [1], which calls for two service providers with different business models (fee-based versus advertising-driven) to share some physical infrastructure, including APs and backhauls. Pricing and service offerings of one company might be determined in part by access statistics that are gleaned from the other.

Problem 3: Apparent network usage. This scenario is possible because various 802.11 packets and fields leak information about network usage. For example, a sequence number field is typically incremented by the sender for each packet that is transmitted. This enables a small number of packets from the AP to be used to gauge the rate of packet transmissions. In beacon frames, which are typically sent by the AP ten times per second, there is a traffic indication map for clients using power-save functionality (which is becoming increasingly popular as a means of extending battery life on small devices). This allows the number of power-save clients to be readily counted, the size of the overall client pool to be estimated from statistics on power-save usage, and perhaps even the length of client sessions to be gauged. These techniques allow Bar to estimate AP usage by observing a relatively small number of packets; any observer can do likewise since these fields are not encrypted. Of course, sampling will always allow a subset of packets to be observed to estimate the whole. Our point is that the current design of 802.11 makes sampling strategies especially effective because 802.11 packets leak much more network usage information than is necessary.

3 Research Challenges

Wireless privacy means more than concealing the contents of a wireless communication, but a precise definition and good metrics that the community can agree upon are elusive because the situation is complicated. Our notion of privacy changes according to who might

be listening. A user might willingly reveal his identity to his provider, but not to other users. And the targets of privacy attacks may range from individuals and their devices to whole companies. Moreover, though several methods have been proposed to measure privacy, such as anonymity sets [5], entropy [9], and k-anonymity [24], there's no agreed-upon threshold for being private enough.

The scenarios presented in the previous section underscore the importance of protecting privacy and highlight several challenges: Ferris was uniquely identified through his ZuNod's MAC address, Boris's clients were revealed in SSIDs, and Bar studied FooNet's operations by monitoring sequence numbers and other information sources. This section discusses three technical challenges immediately relevant to these scenarios: First, how can we balance the need for names to address devices with the desire to prevent names from being identifying? Second, how can we discover and bind to resources without revealing that we are doing so or have done so in the past? Third, might even the physical characteristics of transmissions and the control information contained within them leak other subtle and implicit identifying information, and if so, do we have any hope of designing media access protocols that preserve privacy completely? While we phrase this discussion in the context of our 802.11 case study, we believe that these challenges are broadly applicable to other wireless protocols as well, such as Bluetooth, Zigbee, and WiMAX.

Naming. Network addresses identify communicating endpoints. If they are persistent, they can be used to link multiple packets to the same user. 802.11 uses unique MAC addresses that do not change over time and that are broadcast in the clear. Several potential approaches reduce the threat protocols like 802.11 pose to a user's anonymity, but all increase complexity and computation overhead.

Periodically changing MAC addresses, effectively creating temporary pseudonyms as proposed by Hu and Gruteser [11, 15], would increase the difficulty for both users and service providers to link packet transmissions to a source. In 802.11, this would require only user-level changes to the client, so long as the period between changes is large. For example, a user-level script could change a client's MAC address before each AP association. This approach could be extended to change addresses more often and while associated. The client could generate a new MAC address, re-associate with the AP under that address, and send a gratuitous ARP to establish a binding between his IP address and the new MAC address.

A client might want to use this technique to generate a new pseudonym for each frame it transmits to improve anonymity, but this might be infeasible without first making significant changes to the network stack, as throughput would decrease due to temporarily undeliverable packets and additional messaging. Luckily, many common types of wireless traffic, HTTP for example, are short-lived and spaced out over time, so a user could improve his experience by rolling MAC addresses only when his device is idle; of course, the network he uses will still lose bandwidth to the ARP messages he sends.

However, the risk when using pseudonyms is that they can be linked together, and this is achieved easily when some information carried on a client's transmissions remains constant while his pseudonyms are changing. For example, a client's IP address can be used to link the previous MAC address to a current one. A confidentiality scheme such as WPA2, which encrypts all link-layer data payloads, including ARP messages, can be used, but even this would hide only MAC-to-IP bindings from adversaries who lack network privileges; any eavesdropper that is associated with the wireless network can see gratuitous ARP responses and can send ARP requests. As another example, many machines (*e.g.,* any machine with iTunes music sharing enabled) now enable multicast DNS and DNS service discovery; thus eavesdroppers can now use local-link DNS requests to discover machine name to IP bindings [8]. An effective pseudonym scheme would require coordination across network layers, such as by synchronizing name changes. Note that a consistent identifier is not strictly necessary to link pseudonyms as the sudden cessation of one address and use of another may be suggestive, especially when bolstered by other physical layer characteristics such as signal strength. We explore this non-naming related tracking later.

As well as considering naming and information leakage *vertically* up the network stack, effective privacy solutions must consider leakage *horizontally* across network interfaces and devices. Since many user devices now have multiple radios, such as 802.11 and Bluetooth, an eavesdropper can leverage the persistence of any name on one interface to link the changing names of another. This problem, of course, extends to the multiple radio interfaces of the multiple devices a user might be using simultaneously.

A better approach to hiding persistent identifiers might be to encrypt the addresses, perhaps with a nonce so that successive encrypted identifiers would not be identical. If public or pair-wise shared keys are used, addresses could be hidden even from other authorized users of the network. Unlike the pseudonym approach, however, encryption alone would not prevent a client's communicating peer from learning its identity. Other cryptographic approaches would be needed to provide this added privacy. It would be challenging to adapt cryptographic schemes to the task of address obfuscation, without requiring significant changes to existing media access protocols and without incurring excessive overhead.

Discovering resources and binding. Our wireless de-

vices rely on the ability to discover and bind to services on the fly. We consider several privacy goals relevant to this process, which are challenging to meet. First, only clients who are authorized to use a private service should be capable of learning of its presence. The presumption is that an authorized client would know *a priori* of its existence. Second, at most the client and the service involved should know when a binding is established or broken between them; optionally, the identity of the client may be hidden from the service as well. This would prevent adversaries, such as competing providers, from learning information about how and when a service is used. It is evident from Section 2 that today's 802.11 implementations reveal this information during discovery and binding as they leak SSIDs. Obscuring SSIDs in a simple way, such as by hashing them, might render them unreadable, but they would still be consistent, and could therefore be profiled or mapped offline. Third, a solution that provides private resource discovery and binding should be secure from common attacks, such as man-in-the-middle, spoofing, and replay, as well as be compatible with existing media access protocols.

To achieve these goals in 802.11, one might design an anonymous messaging scheme in which the contents of all management frames used for service discovery, authentication, and binding (*i.e.,* association) are encrypted with either public or shared keys; moreover, packet lengths and remaining cleartext fields, such as the frame types, could be made homogenous. However, a comprehensive design additionally would need to deal with several serious challenges that arise in practice. These include: bootstrapping cryptographic state at autonomous clients; disseminating and consistently managing such state at APs; ensuring system scalability as the number of clients increases; synchronizing cryptographic state between a client and an AP in the face of message loss caused by wireless links; ensuring that the encryption operations do not leak the identities of the intended recipients (since standard public key encryption schemes need not provide key-privacy [4]); and making sure that the resulting scheme has acceptable overheads.

As a strawman approach, consider a client who knows the public keys or identities of all the APs to which it may wish to associate. The client could send an encrypted probe to each AP using an *anonymous* public key [4] or *anonymous* identity based [2] encryption scheme. The probe could include a (possibly ephemeral) public key for the client, and the target AP could use this public key to encrypt the response. This strawman approach is computationally heavy for both the client and the APs; the anonymity property of the encryption schemes means that the APs must perform non-trivial computations on each encrypted probe, even if the probe is intended for a different AP. Our strawman approach shares commonality with the randomized hash lock protocol for anonymous authorization [26] in which an RFID tag reader must try all tag keys in order to determine the identity of an RFID tag. To improve upon this strawman, one might consider amortizing the cost of expensive cryptographic operations over multiple sessions, albeit with a potential degradation in the level of privacy provided. For example, after binding with a resource once, a client might maintain a single use or time-limited token that would efficiently catalyze the client's next attempt to discover that resource. As in [15], the token itself may be blinded to prevent the AP from linking two clients together over time.

Limiting information leakage. Explicit names aren't the only identifiers. Other information conveyed in a wireless link layer protocol might reveal clues to a user's identity to varying degrees. The challenge is to find such *implicit identifiers*, measure their utility, and when necessary, devise strategies to remove them while retaining useful functionality and without noticeably degrading performance.

802.11 frames reveal sequence numbers, operating modes, and capabilities to anyone listening, which can be used to link multiple packets to the same source as well as to narrow down the range of the transmitter's possible identities. This information is found in frame headers and control and management frame payloads, but even the best confidentiality schemes cover only the payloads of data frames. An obvious privacy solution would be to encrypt the entirety of every type of frame, but doing so might not be as easy as it seems. First, some link header fields are designed to be broadcast to all users. For example, the duration field is used to announce to all contending users how long the channel will be in use. Encrypting such a field might require using a key that is shared by all authorized users, and would thus be defenseless against an attack by an authorized user; FooNet and other metropolitan-scale networks have thousands of authorized users, thus sharing a key among them wouldn't provide much protection. Second, if a client were to encrypt the remaining fields so that only the AP could decrypt them, then the AP would suffer additional computation load, and would thus be more susceptible to denial-of-service attacks.

Encryption, moreover, might be insufficient. Wireless protocols are often defined to support a number of configuration parameters, be extensible, and provide slack. This gives device manufacturers, driver developers, and end-users a wide range of usage options. In general, this configurability encourages innovation, reduces costs, and improves performance, as communication can be tailored to a particular environment or application in use. However, when a device uses a protocol in a slightly different way—exercising some options and not others—it makes it easier to profile and fingerprint it. For example, we easily can differentiate a device that uses power-save mode or virtual carrier sensing (RTS/CTS) from one that never

does. Likewise, it is easy to tell apart two devices that send background probe requests every 60 and 60.2 seconds respectively.

Identifying information may also be found in protocol timings, system and network card clock skew, radio frequency fingerprints, how the client chooses a data transmission rate in response to prevailing conditions, data transmission timings (e.g., the periodicity of probe requests), how the received signal strengths of multiple packets can be linked to the same sender, and how packet lengths are distributed. This information may be found in single packet transmissions, as well as by profiling a group of packets that are presumed to be from the same source. Note for the latter, to ensure privacy, it might be sufficient only to obscure the linkings of packets (to prevent grouping them in the first place). Although some work has demonstrated that these leaks may be used to identify which device driver is in use [10] or otherwise fingerprint a device [17], it remains to be seen whether they provide enough identifying information to profile users uniquely. If they do, then the challenge is not only to plug them, but to do so efficiently. The remedies against timing attacks, packet length profiles, and signal strength correlation might be to induce delays, pad packets, and adjust transmission powers; unfortunately, these solutions would all result in reduced throughput.

4 Conclusions

This paper argues that wireless networks pose new threats to privacy and that there is incentive, not just for users, but also for providers and manufacturers to address these threats. In an effort to explain how to make these networks private, we explain how one might identify the presence of particular users or types of devices, both by looking at explicit identifiers such as names and addresses which are used to discover networks and deliver data, and by profiling users' communications to develop implicit identifiers. Furthermore, we argue that prior efforts to anonymize communication still leak explicit identifiable information during service discovery. It seems that many research challenges remain to be solved before Ferris can have his day off!

References

[1] EarthLink and Google win San Francisco Wi-Fi bid. `http://news.com.com/EarthLink+and+Google+win+San+Francisco+Wi-Fi+bid/2100-7351_3-6058432.html`.

[2] M. Abdalla, M. Bellare et al. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. In *Crypto'05*, 2005.

[3] P. Bahl and V. N. Padmanabhan. Radar: An in-building RF-based user location and tracking system. In *INFOCOM*, 2000.

[4] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval. Key-privacy in public-key encryption. In *ASIACRYPT'01*, pages 566–582. Springer-Verlag, 2001.

[5] A. R. Beresford and F. Stajano. Location privacy in pervasive computing. *IEEE Pervasive Computing*, 2(1):46–55, 2003.

[6] A. Biryukov, A. Shamir, and D. Wagner. Real time cryptanalysis of A5/1 on a PC. In *FSE '00: Proc. 7th International Workshop on Fast Software Encryption*, pages 1–18. Springer-Verlag, 2001.

[7] N. Borisov, I. Goldberg, and D. Wagner. Intercepting mobile communications: The insecurity of 802.11. In *Proc. MobiCom*, pages 180–189, 2001.

[8] S. Cheshire and M. Krochmal. Multicast DNS, Aug 2006. URL `http://www.multicastdns.org/`. IETF Draft.

[9] C. Diaz, S. Seys, J. Claessens, and B. Preneel. Towards measuring anonymity. In *Workshop on Privacy Enhancing Technologies*, volume 2482 of *LNCS*, 2002.

[10] J. Franklin, D. McCoy et al. Passive data link layer 802.11 wireless device driver fingerprinting. In *Proc. 15th USENIX Security Symposium*, pages 167–178, jul-aug 2006.

[11] M. Gruteser and D. Grunwald. Enhancing location privacy in wireless LAN through disposable interface identifiers: A quantitative analysis. *Mob. Netw. Appl.*, 10(3):315–325, 2005.

[12] M. Gruteser, G. Schelle, A. Jain, R. Han, and D. Grunwald. Privacy-aware location sensor networks. In *HotOS IX*, 2003.

[13] U. Hengartner and P. Steenkiste. Access control to information in pervasive computing environments. In *HotOS IX*, 2003.

[14] U. Hengartner and P. Steenkiste. Access control to people location information. *ACM Trans. Inf. Syst. Secur.*, 8(4), 2005.

[15] Y.-C. Hu and H. J. Wang. A framework for location privacy in wireless networks. In *Proc. ACM SIGCOMM Asia Workshop*, April 2005.

[16] A. Juels. RFID security and privacy: A research survey. *IEEE Journal on Selected Areas in Communication*, 24(2), Feb. 2006.

[17] T. Kohno, A. Broido, and k. claffy. Remote physical device fingerprinting. In *IEEE Symposium on Security and Privacy*, pages 211–225, May 2005.

[18] A. LaMarca, Y. Chawathe et al. Place Lab: Device positioning using radio beacons in the wild. In *Proceedings of Pervasive 2005*, 2005.

[19] J. Li, C. Blake, D. S. D. Couto, H. I. Lee, and R. Morris. Capacity of ad hoc wireless networks. In *MobiCom '01: Proc. 7th annual international conference on Mobile computing and networking*, pages 61–69. ACM Press, 2001.

[20] Microsoft. Wireless client update for Windows XP with service pack 2. URL `http://support.microsoft.com/kb/917021`.

[21] Muniwireless. URL `http://www.muniwireless.com/`. MuniWireless is the portal for news and information about citywide wireless broadband projects around the world.

[22] G. Myles, A. Friday, and N. Davies. Preserving privacy in environments with location-based applications. *IEEE Pervasive Computing*, 2(1), 2003.

[23] E. O'Neill, V. Kostakos et al. Instrumenting the city: Developing methods for observing and understanding the digital cityscape. In *Ubicomp*, pages 315–332, 2006.

[24] L. Sweeney. k-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, 2002.

[25] R. Want, A. Hopper, V. F. ao, and J. Gibbons. The active badge location system. *ACM Trans. Inf. Syst.*, 10(1), 1992.

[26] S. A. Weis, S. E. Sarma, R. L. Rivest, and D. W. Engels. Security and privacy aspects of low-cost radio frequency identification systems. In *Security in Pervasive Computing*, volume 2802 of *Lecture Notes in Computer Science*, pages 201–212, 2004.

[27] Wigle. URL `http://www.wigle.net/`. Wireless Geographic Logging Engine: Making maps of wireless networks since 2001.

[28] F.-L. Wong and F. Stajano. Location privacy in Bluetooth. In *ESAS*, pages 176–188, 2005.

Auditing to Keep Online Storage Services Honest

Mehul A. Shah, Mary Baker, Jeffrey C. Mogul, Ram Swaminathan (HP Labs, Palo Alto)
Mehul.Shah@hp.com, Mary.Baker@hp.com, Jeff.Mogul@hp.com, Ram.Swaminathan@hp.com

Abstract

A growing number of online service providers offer to store customers' photos, email, file system backups, and other digital assets. Currently, customers cannot make informed decisions about the risk of losing data stored with any particular service provider, reducing their incentive to rely on these services. We argue that third-party *auditing* is important in creating an online service-oriented economy, because it allows customers to evaluate risks, and it increases the efficiency of insurance-based risk mitigation. We describe approaches and system hooks that support both *internal* and *external* auditing of online storage services, describe motivations for service providers and auditors to adopt these approaches, and list challenges that need to be resolved for such auditing to become a reality.

1 Introduction

The year is 2027. Alice and Bob are preparing a family photo album for the wedding of their daughter Carly, and want to include her baby photos. They took these photos with a digital camera in 2002, and uploaded them to the LensElephant photo-sharing service. For 25 years, they have paid LensElephant a small fee to store their photos. But today, after they click on Carly's baby-photo thumbnails to view the full files, they see nothing but an error message "Sorry, those files are corrupted."

This scenario is not just a dark fantasy. Many online services allow customers – both end users and businesses – to store data for as long as they want, and some charge for the service. Yet news reports [1, 5] reveal that even the most popular sites can lose data, and customers have no rational basis on which to evaluate the risk of data loss or to choose between storage services. Although examples in this paper focus on online storage services, the problem generalizes to the nascent "online service-oriented economy" (OSOE) in which businesses and end users purchase IT services from a variety of online service providers (OSPs). For this nascent economy to become established, customers will need ways to assess risk and gain trust in OSPs.

Third-party auditing is an accepted method for establishing trust between two parties with potentially different incentives. Auditors assess and expose risk, enabling customers to choose rationally between competing services. Over time, a system that includes auditors reduces risk for customers: when combined with a penalty or incentive mechanism, auditing gives incentives to providers to improve their services. Penalty and incentive mechanisms become supportable when risks are well understood. We believe auditing is necessary not only for traditional businesses, but also for online services to create a viable OSOE.

Auditing of OSPs is not feasible yet. First, customers are not yet sophisticated enough to demand risk assessment. Second, OSPs do not yet provide support for third-party audits.

In this paper, we describe the issues involved in making auditing online services a reality. We speculate on an insurance-based incentive system to encourage audits of OSPs. For concreteness, we focus on audit-enabling interfaces that online storage services should support.

2 What We Mean by Auditing

OSPs must convince customers that their platforms are reliable. Because a customer knows that a provider's primary incentive is to make a profit, not simply to serve the customer's needs, this is a variant of what economists call the "principal-agent problem" [14]: how does the customer know whether to trust the provider?

One way is to rely on a trusted third-party auditor, who has sufficient access to the provider's environment. An auditor understands the service level agreement (SLA) between a customer and a provider and quantifies the extent to which a provider might not meet the SLA. The auditor has expertise and capabilities that the customer does not. Auditors understand the threats posed, know best practices, and have the resources to check for process adherence and service quality. They perform these checks through well-defined interfaces to the service. With proper safeguards, auditors can investigate providers who serve multiple customers without fear of information leakage.[1]

Inspired by methods for auditing "bricks and mortar" businesses, we identify two approaches: *external auditing* and *internal auditing*. We also suggest some principles that should apply to auditing of online services.

2.1 Internal vs. external auditing

We illustrate the two types of audits using an analogy to the restaurant business. A restaurant's "SLA" is to provide its diners with enjoyable and safe meals at reasonable expense. We can audit restaurants simply by trying out the service, or (using third parties) by relying on recommendations from friends or food critics. This corre-

[1] This paragraph is adapted from [9].

sponds to *external* auditing of a service. External audits evaluate the quality of service through externally available interfaces, and usually assume that one can predict future success ("reputation") from limited samples. With this approach, we can determine whether the restaurant is enjoyable, but it might not warn us in advance if the food is sometimes contaminated.

So we also rely on health inspectors to go into the kitchen and identify procedures that might result in sick customers. This corresponds to *internal* auditing, which determines the extent to which a service follows best practices. Internal audits evaluate the structure and processes within a service to ensure that the service can continue to meet its objectives (SLAs). To do so, these audits need specialized interfaces that reveal and test the inner workings of a service. Other examples of internal audits include fire marshal inspections of office buildings to check electrical codes, and accounting firms checking whether corporations meet FASB standards. As with health inspections, these internal audits warn customers of practices that could lead to impending disasters. Customers can use this information to choose between services or complement one service with another.

We need both internal and external audits of OSPs. External audits can only confirm past behavior, so without internal audits, we could not predict upcoming problems or assess risk exposure. On the other hand, internal audits might not be exhaustive and might be based on incomplete failure models; we can use external audit results to assess whether internal audits are really working.

2.2 Auditable metrics

Auditing requires metrics. Typically, best practices determine the relevant metrics for internal audits, and the SLAs determine the relevant metrics for external audits. Metrics need not be quantitative; they may simply indicate whether the service follows a certain practice. For example, a financial audit might (among other metrics) check whether accounting data conforms with appropriate standards; a health inspection could verify that raw food is always stored below 45°F. Sections 4 and 5 describe metrics for internal and external audits of storage services, respectively.

2.3 Desirable properties

In this section, we list desirable properties for both internal and external audits. These are ideal goals, and might not all be fully achievable.

Establish standards for comparison. One purpose of auditing is to allow customers to make a rational choice. To do so, they need standards to compare the results of two different audits performed at different times, of different providers, and by different auditors. Without standards we are not even sure what is being audited.

Minimize auditing cost. Auditing imposes costs on OSPs, which should not outweigh its benefits.

Introduce no new vulnerabilities. Auditing interfaces inherently provide new ways to access an online service. These interfaces, and their implementations, should not introduce ways of attacking or corrupting the system.

Protect customer data privacy. Customers often do not want their data revealed or leaked. For OSPs that include customer privacy in their SLA, auditing approaches that reveal customer data to auditors are a vulnerability and should be avoided.

Protect proprietary provider information. OSPs seldom want to reveal internal information (including key technologies) to competitors or hackers.

Audit results must be trustworthy. Audits should not be biased either toward the OSP or the customer.

Avoid prescribing a technology. Technologies used to implement and support online services continually change. Audit practices that prescribe or prefer a particular technology could inhibit transitions to better technologies. Thus, external audits should rely on implementation-independent interfaces, if possible. Although internal audits must be technology-aware to spot flaws in technical approaches, they should adapt quickly to avoid constraining service implementors.

3 Motivating audit processes

Providers will not offer auditing interfaces unless there is motivation to do so. Mechanisms to provide such motivation are more likely to be social than technical, but we should keep them in mind when trying to design the system interfaces that support auditing.

Generally, these behavior-changing mechanisms either use penalties or incentives, or a combination. For example, regulations (and the associated fines), laws (and the threat of incarceration), or loss of reputation (which can put a provider out of business) are penalty-based mechanisms. Market forces (i.e., the ability to charge a premium for better service), or the need to obtain cost-effective insurance, can create incentives.

3.1 The role of insurance providers

Users of traditional services (e.g., home remodeling contractors) understand the value of selecting insured providers. Similarly, we believe that in many cases customers are likely to prefer insured OSPs, since it gives them both protection against SLA violations and a clear signal about which providers are trustworthy enough to be insurable.

Insurance companies will not insure OSPs unless they can quantify their risk, which gives both insurers and OSPs an incentive to use auditors. Since insurance premiums will reflect both the known risks and the uncertainty in risks, OSPs will have an incentive (reduced premiums) to improve their risk-management practices and to increase their transparency to auditors. Insurers will pass audit costs to the OSPs, giving them an incentive to improve auditing interfaces to reduce audit costs.

Others have projected this benefit for a more specific area, security-risk auditing:

> Beginning in January 2007, "financial service providers [in South Korea] will be required to insure customers' accounts to cover financial damage caused by hackers and financial accidents." ... [this] will, in effect, turn insurance providers into [security] compliance auditors. Those financial services providers who follow sound security procedures will be a better risk, and therefore see more favorable insurance rates. [7]

In theory, individual consumers could do their own external audits, but they are unlikely to be able to afford the costs and skills associated with professional risk audits. Insurance-based mechanisms are scalable; they allow an OSP to efficiently convince many customers that their SLAs will be met.

Insurance-based mechanisms are especially suitable for online storage services. We usually protect valuable physical assets (e.g. homes, jewelry) with insurance, so it is natural to extend this mechanism to digital assets. For this mechanism to work, however, we need ways of assigning value to data, which remains an open problem. For certain data, such as business assets, we may be able assign a value based on the cost incurred as a result of loss. However, this assessment is more difficult for sentimental data such as family photos.

Although insurance-based incentives may not extend to all online services, they are applicable to many OSPs, such as financial or utility computing services.

3.2 Chicken-or-egg Problem

We realize that existing OSPs will face costs to add support for auditing, both internal and external. These include additional hardware, new software, additional network bandwidth, new procedures, additional employees, and training. There is also a chicken-or-egg problem: OSPs will be reluctant to incur these costs before auditing is the norm, but auditing cannot become ubiquitous until the OSPs support it.

We see a plausible evolutionary path, however. OSPs will initially perform self-auditing to comply with regulations such as Sarbanes-Oxley (SOX); most large companies already do significant IT auditing for SOX compliance, although they probably do not yet check long-term storage processes. Once these self-audit mechanisms are in place, they can then be used as selling points, to differentiate well-run providers against their competition. Insurance companies will then also have a basis for writing data-loss policies.

3.3 Quid custodiet ipsos custodes?

Auditing depends on the competence and honesty of the auditors, properties that themselves need to be enforced. Financial auditors are already subject to regulatory and legal sanctions (for example, Arthur Anderson was forced to close its audit business after repeated instances of malfeasance, including their work for Enron). Insurers will also tend to avoid auditors who do bad jobs, as this exposes the insurers to unprofitable risks.

Likewise, insurance-based incentives depend on the competence and honesty of the insurers. We assume that OSOE insurers will be subject to existing insurance-industry regulations, and will not disappear when it comes time to pay up.

Having multiple competing auditors can help ensure trustworthy results. This implies the need for standard auditing interfaces, to avoid "auditor lock-in."

4 Internal Auditing for Storage Services

Internal auditing checks the inside behavior and processes of a provider to assess the likelihood the provider will fail to meet its SLAs. In this section, we describe the difficulty in internally auditing storage services and outline auditing interfaces online storage services can support to address the difficulty.

The SLAs for a storage service can include data integrity (customer bits are preserved), data exit (customers can get their original data), security, privacy, and more. Although auditing can apply to all of these, in this paper we mainly focus on data integrity.

4.1 Internal audits are hard

To assess the risk of violating data integrity, an auditor must understand the threats and the current best practices to prevent data loss. Unfortunately, this information is hard to obtain for storage services:

- Only a few studies describe failures and their causes in deployed large-scale storage systems [10, 12]. Baker *et al.* [2] provide a list of known threats to data preservation, e.g. natural disasters and insider attack.
- Individual organizations can learn to recognize certain bad practices that have resulted in data loss, but determining "best practices" requires comparison and experimentation. This in turn requires sharing failure information across different environments and vendors. Storage services have been reluctant to share this data.

The current situation is counter-productive for both customers and providers. Customers cannot assess service quality, and storage services can only make improvements based on internal data. Once a system with proper incentives is established, storage services will allow internal audits. Trusted third-party auditors will then be in a position to piece together this information gradually and integrate it into the audit process.

4.2 Threats and hooks for internal audits

We list three classes of threats to data integrity that audits must address. (There are others!) For each class, we touch on potential best practices and outline the hooks storage services could provide to check for adherence to these practices.

Latent faults: Many potential sources of data corruption are not immediately visible (e.g., damage from suc-

cessful but undetected attacks; undetected human errors; bit rot in the storage medium). These "latent" faults require periodic checking of the data [2]. Auditors need hooks to know whether the storage service checks for appropriate sources of latent faults, whether the checks happen often enough, and whether there are enough independent replicas of the data to support successful repair when the checks uncover damage. To support auditing, the storage service could provide interfaces that access log files to verify the frequency and success of checks. Another possible hook would allow an auditor to implant triggers on random data to notify the auditor when the data is accessed for particular checks.

Correlated faults: Correlated failures increase the risk of data loss. Example sources of correlated failures include lack of diversity in software and hardware platforms, in geographic locations of storage, and in number of independently administered domains. Storage services should provide hooks that expose the level of diversity along these and other dimensions, to gauge the potential for correlated failures.

Recovery faults: Data is often more vulnerable to corruption and loss during recovery procedures, since these might be executed only rarely and are often less well debugged. Recovery also tends to take effect after things have gone wrong, when the system being recovered may be in an unanticipated state. Thus, it is important for storage services to practice regular "fire drills," (in which they deliberately disable disks, nodes or sites or deliberately corrupt or destroy data), and then measure the success of the resulting recovery mechanisms. System hooks should allow auditors to see logs of these faults and the resulting recovery activity. (Faulty recovery may corrupt unrelated data, which might only be detected through further audits.)

More importantly, storage services should provide hooks that allow auditors to test recovery safely. They should allow auditors to inject data corruption into "decoy" files, or to take disks or systems offline temporarily.

There are many open issues related to internal auditing, including:

- How much overhead does internal auditing add?
- Could internal audits cause unwanted damage?
- Beyond providing a checklist of preventive steps taken, we do not have ways to translate internal audits into a quantitative risk of *future* data loss.
- We will need ways to evolve the internal-auditing process and hooks without disrupting storage services.

5 External Auditing for Storage Services

An external audit provides an end-to-end measurement of service quality in terms of its SLA. The primary SLA for storage services is to maintain data integrity, which we quantify by the *past* rate of data loss. We obtain this rate by measuring the fraction of data lost at appropriate intervals. There are a few direct methods for measuring this fraction, such as sampling the service's stored contents or simply downloading all content. External sampling cannot detect small but important losses because it only checks a fraction of the data. Exhaustive checks are intrusive and infeasible. In this section, we detail the problems with these approaches and propose a solution to overcome these problems.

5.1 Hurdles for detecting data loss

A simple method for auditing data integrity is to sample stored data through the public read and write interfaces of a storage service. This approach requires no modification to the service. The auditor simply creates some "fake" user accounts, uploads content, and periodically extracts all of the uploaded content and ensures the result matches the original. For some services (e.g., Amazon's S3), we can access the read and write interfaces programmatically. For other services (e.g., email or photo sharing sites), we might need additional effort (e.g. screen-scraping) to crawl and download uploaded content. We cannot externally audit services that actively prevent users from downloading their content. If our premises about the advantage of auditing are correct, customers will eventually avoid such services.

This simple approach meets our guidelines but has drawbacks. First, devious services would have an incentive to identify auditors' accounts and devote more resources to those than others. Second, the approach suffers from limited coverage. Even if the service is not biased, the auditor can only introduce a limited number of fake accounts and a limited amount of data without excessive overhead to both service and the auditor. A small sample can only detect loss from failures that affect a large fraction of the data (e.g. natural disasters or major operator error).

Another option is for storage services to provide dedicated, auditor-only read interfaces that allow access to all internal customer data. The auditor can then compute cryptographic hashes (e.g., using SHA-1) of each customer data object, and then replicate these hashes (perhaps at other storage services), later using these to check the original content. Unfortunately, to detect small amounts of loss, an audit must check nearly all the data. For example, imagine an SLA that requires a storage service to maintain 1PB, and the service has lost ten 1KB blocks. If no additional damage occurs, we would need to randomly sample 40% of the 1PB to have better than a 98% chance of detecting at least 1 lost block. Since auditors will want to know the loss rate with high precision, this approach is too expensive. It requires transferring nearly all customer data, which would consume too much network bandwidth. This approach also violates our customer-privacy principle. Straightforward encryption is not a viable privacy solution, since it relies on

keeping secret keys for the long term. Customers are prone to losing keys, and the auditor has no way to check if the key remains intact.

5.2 A privacy-preserving approach

We propose a solution that allows auditors to detect any change to stored data, malicious or otherwise, while adhering to our auditing principles. In our scheme, we encrypt data to hide it from the auditor, and we store *both* the encrypted data and key with the storage service. To check data integrity, our method uses a challenge-response protocol in which the service can respond correctly only if both the encrypted data and key are intact. The auditor can glean nothing about the data or key from these checks, and these checks incur little network overhead since the messages are small. Our solution also allows the auditor to assist in returning the key and encrypted data to the customer, while maintaining customer privacy. With our method, the customer need not maintain any long-term secrets or state, and the auditor needs only a little long-term state.

Our scheme is based on the following assumptions. Since many sites, such as email hosting or photo sharing, offer value-added services beyond storage, customers are willing to reveal their data to a service. Services have other legal and social incentives not to leak this data. A customer may claim data loss either innocently or fraudulently, and services have an incentive to hide data loss. The auditor does not collude with either the service or customer, because its task is to assess service quality accurately. The auditor can, however, accidentally or purposefully gain or leak private customer information.

Our solution has three phases: *initialization*, *verification*, and *extraction*. In initialization, the storage service commits to storing a document on behalf of the customer, and the auditor initializes long-term state. During verification, the auditor repeatedly checks the stored contents. Finally, in extraction, the auditor assists in returning the data to the customer in case of doubt or dispute. We sketch each of these next.

During initialization, the storage service commits to storing the key, K, and encrypted data, $E_K(D)$, after receiving these items from the customer. The service commits by publishing two values: a *data-commitment* that is a hash of the encrypted data, $H(E_K(D))$, using a secure digest such as SHA-1, and a *key-commitment* that hides the key using modular exponentiation, $g^K \bmod p$ (g is a generator of Z_p^*, and p is a large prime). Because these functions are one-way and collision-resistant, these values bind the service to the encrypted data and the key without revealing either. The service publishes these values to one or more auditors. (We also have methods to transfer the values from one auditor to the next, or make them publicly available so anyone can audit.) An auditor must remember these values, typically much smaller than the data, for all subsequent checks.

The auditor also precomputes a list of challenges and corresponding responses for checking the encrypted data. Each challenge-response pair is a random number, R_i, and a hash (implemented with keyed-hash message authentication codes — HMACs) that can be computed only by knowing R_i and the encrypted data in entirety. This list is much smaller than the encrypted data, and its contents are unknown *a priori* to the service.

During verification, the auditor must check that (a) the encrypted data is unchanged and (b) the encryption key is unchanged. For (a), the auditor randomly selects a pair from its precomputed list, sends the challenge to the service, and awaits the correct response. The auditor must discard the pair after use, otherwise the service may cheat by using previously cached results. When the auditor exhausts this list, it can generate more pairs by getting the encrypted data from the service, which is potentially expensive, and verifying the encrypted data's hash.

Our main innovation is how to repeatedly check (b) without revealing the key to the auditor. We rely on the hardness of the discrete-log and the commutativity of modular exponentiation to generate challenge-response pairs from the key-commitment. The key-verification protocol between the auditor, A, and service, S is:

1. A chooses a random β s.t. g^β is a generator of Z_p^*.
2. A sends g^β to S.
3. S computes and sends $(g^\beta)^K$ to A.
4. A checks $(g^K)^\beta = (g^\beta)^K$ else declares S **lost** key.

In steps 1-2, the auditor generates a random challenge while keeping β secret from the service. Because of the hardness of the discrete-log, the service, in step 3, must compute the response anew using the key. Since modular exponentiation commutes, the auditor uses the key-commitment, in step 4, to check the response.

For extraction, the auditor assists in returning the encrypted data simply by forwarding it to the customer after verifying its hash. The auditor also forwards the key, but before forwarding, we must first establish a shared random secret, X, between the service and customer, C, which is used to hide the key. The auditor also needs g^X to check the key. We skip the protocol for pre-arranging these values and instead focus on key forwarding:

1. S sends $K + X$ to A.
2. A checks $g^K g^X = g^{K+X}$ else declares S **lost** key.
3. A sends $K + X$ to C.

In step 1, the shared secret, X, is a "blinding" value that hides the key. In step 2, g^X hides the blind from the auditor, while allowing it to verify the key. The customer, after step 3, extracts the key by subtracting X.

Our protocols detect data loss and are not vulnerable to a cheating storage service. Given an honest auditor, the protocols either leave no doubt that the key and data are intact or reveal which party is at fault. We can prove

that the auditor, honest or otherwise, learns nothing about the data contents when using these protocols. Thus, the auditor can reveal transcripts of the interaction to additional external parties, thereby providing an audit trail of the audit itself, without violating our privacy principle.

To support our protocols, storage services must export hooks for challenge-response queries and compute expensive functions for responses. Since our protocols mostly send small hashes, the main overhead comes from computing HMACs rather than network traffic. If we limit the number of checks, however, this overhead can be tolerable for a service. We measured the performance of a SHA-1 HMAC over files stored on five 500GB SATA disks with a 2-core 2GHz Intel Xeon 5130 at 362 MB/s. At this peak rate, 50 CPUs in parallel can check 1PB in 16 hours. Spread over 30 days, this work imposes less than 2% overhead. Since many large-scale storage services handle an archival workload in which most data is rarely touched, checking monthly is reasonable.

6 Related Work

Storage-related auditing: Baker *et al.* [2] describe threats to long-term storage and a reliability model, which includes the effect of latent faults and periodic internal checks of data integrity. Miller and Schwarz [13] describe an efficient method for checking online storage, but it cannot provide full coverage and privacy simultaneously. OceanStore [4] periodically checks stored data for integrity, but relies on users to keep secret keys for privacy and does not offer an interface for external auditing. LOCKSS [8] is a P2P archival system for library periodicals. Although a library site in LOCKSS periodically performs audits of its content, there is no need for complete privacy since the contents are published. Lillibridge *et al.* [6] present a P2P backup scheme in which peers request random blocks from their backup peers. This scheme can detect data loss from free-riding peers, but does not ensure *all* data is unchanged.

Auditing in general: One of us (Mogul) previously suggested that auditing support would become necessary for IT outsourcing in general [9]. Satyanarayanan proposed dealing with a variety of "Internet risks" by an audit-like mechanism ("inspection-enforced safety") based on periodic signed, encrypted snapshots of entire virtual machine states, which would then be inspected when necessary (e.g., during legal proceedings) [11]. Others have suggested building accountable systems that provide an non-repudiable history of their state and actions [15]. These audit trails are useful for detecting and pinpointing problems after the fact, and could support internal and external audits.

Click fraud scares businesses that advertise through search engines [3]. To preserve advertiser confidence (on pain of lost business), search engines internally self-audit click streams; also, advertisers can externally audit search engines by monitoring the "click-throughs."

7 Conclusion

In this paper, we motivate the need for auditing to support an online service-oriented economy. We highlight issues around both internal and external auditing and detail ways of auditing online storage services.

References

[1] M. Arrington. Gmail Disaster: Reports of Mass Email Deletions. TechCrunch, `http://www.techcrunch.com/2006/12/28/gmail-disaster-reports-of-mass-email-deletions/`, Dec. 2006.

[2] M. Baker, M. Shah, D. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A Fresh Look at the Reliability of Long-term Digital Storage. In *Proc. EuroSys*, Leuven, Belgium, Apr. 2006.

[3] B. Grow and B. Elgin. Click Fraud: The Dark Side of Online Advertising. BusinessWeek Online, `http://www.businessweek.com/magazine/content/06_40/b4003001.htm`, Oct. 2006.

[4] J. Kubiatowicz et al. OceanStore: An Architecture for Global-Scale Persistent Storage. *ASPLOS*, Nov. 2000.

[5] D. Lazarus. Precious Photos Disappear. San Francisco Chronicle, `http://www.sfgate.com/cgi-bin/article.cgi?file=/chronicle/archive/2005/02/02/BUG7QB3U0S1.DTL`, Feb. 2005.

[6] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A Cooperative Internet Backup Scheme. *Proc. USENIX Annual Conf.*, pages 29–41, June 2003.

[7] T. Liston. Korean Financial Service Providers Required to Insure Accounts. SANS NewsBites, `http://www.sans.org/newsletters/newsbites/newsbites.php?vol=8&issue=97`, Dec. 2005.

[8] P. Maniatis, D. S. H. Rosenthal, M. Roussopoulos, M. Baker, T. Giuli, and Y. Muliadi. Preserving Peer Replicas by Rate-Limited Sampled Voting. In *Proc. SOSP*, pages 44–59, Oct. 2003.

[9] J. C. Mogul. Operating Systems Should Support Business Change. In *Proc. HotOS X*, June 2005.

[10] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proc. FAST*, Feb. 2007.

[11] M. Satyanarayanan. Inspection-enforced Internet Safety. In *NSF Workshop on Grand Challenges in Distributed Systems*, M.I.T., Sep. 2005. `http://pdos.csail.mit.edu/~kaashoek/nsf/`.

[12] B. Schroeder and G. Gibson. Disk Failures in the Real World: What does an MTTF of 1,000,000 hours mean to you? In *Proc. FAST*, Feb. 2007.

[13] T. Schwarz and E. Miller. Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage. *ICDCS*, July 2006.

[14] J. K. Shim and J. G. Siegel. *Dictionary of Economics*. Wiley, 1995.

[15] A. Yumerefendi and J. Chase. The Role of Accountability in Dependable Distributed Systems. *HotDep*, 2005.

A Web based Covert File System

Arati Baliga, Joe Kilian and Liviu Iftode
Department of Computer Science
Rutgers University, Piscataway, NJ.
{*aratib, jkilian, iftode*}*@cs.rutgers.edu*

Abstract

We present the idea of a web based covert file system, CovertFS. This file system allows a user to store files covertly on media sharing websites while guaranteeing confidentiality and plausible deniability regarding the existence of the files. Further, it allows for selective and covert sharing of these files with other users. CovertFS can be built on top of any web based media sharing service. The files are hidden within the media using steganographic techniques. The user can plausibly deny the existence of the covert file system since the existence of it cannot be proven. The media sharing service provider is oblivious to the existence of the file system within the stored media, providing them plausible deniability as well. Since the user files are completely hidden, it gives only the user complete control over his confidential files.

1 Introduction

Web services such as email, photo sharing, video sharing, blogs, wikis and other collaborative and interactive services have become a part of our daily lives. The web provides an easy and portable means for storing and retrieving user content as well as sharing this content within a group of people for online collaboration.

All web services available today are for open storage and sharing, where the existence of the data is known to the service provider. The fundamental, implicit assumption here, is that the service provider can be completely trusted with the user data. Any content stored in the clear on these servers is vulnerable to unauthorized access by the service administrators. Further, the government could compel the service provider to turn over this data without the knowledge of the user. A more cautious user might encrypt all content that is stored on these servers. While this protects the data from unauthorized access, it cannot hide the fact that some data is stored by a particular user. The user might be subsequently coerced into revealing the encryption keys by legal instruments such as subpoenas. Thus, users may desire to hide the very presence of their data stored on public servers in such a way that its existence cannot be proven by the service providers themselves or another third party.

Storing and sharing data covertly over the internet serves several purposes. For example, this may be used as a means to share content in societies that tend to stifle free exchange of unpopular ideas. Even in more democratic countries, social taboos can force people to look for covert means for facilitating secret online collaborations. Finally, individual web users may use such covert means to backup, store and share their files online without the knowledge of the service providers.

In this paper, we propose the idea of a covert web based file system, CovertFS, which facilitates secure file storage and sharing amongst a group of people and yet provides plausible deniability. CovertFS can be built on top of any publicly available media hosting and sharing service. Flickr [2], a photo sharing service from Yahoo, is an excellent example of such a service as it provides large storage and excellent API. The file system is covertly hidden within the media hosted by the user using steganographic techniques. This file system provides plausible deniability for the user and the service provider. Plausible deniability is achieved because the presence of the hidden data cannot be determined by any external parties, including the service provider.

The salient features of the file system can be stated as follows:

- **Plausible Deniability:** The presence of the file system or files within the media sharing web account cannot be determined with certainty by analyzing the media or the traffic. Hence the user or the service provider cannot be compelled by court to disclose the contents of the file system. This form of information hiding is desired by users who can

safely and securely store their documents on third party servers without the knowledge of the service providers. The service provider cannot determine with certainty whether the media is a plain media file or a media file with hidden content.

- **Online File Sharing and Collaboration:** The file system is built on top of a web based media sharing service. This makes the file system available online and to anyone anywhere to collaborate or share files with one another. The additional benefit of this is only the end-users are aware of the file system organization and contents. To others, the file sharing traffic looks like innocuous media sharing traffic.

- **Information Hiding:** The file system is aimed at hiding confidential documents, which can be stored and shared between a group of people. Data is hidden within the media using advanced steganographic techniques such as secure or zero divergence steganography [13, 12, 11], which cannot be steganalyzed to retrieve the hidden content.

2 Design Overview

The design of a covert file system on a media sharing website poses several research challenges. First, an efficient way to hide the file system data within photos is necessary. Advances in steganography help us here [13, 12, 11]. The file system data can be encrypted and hidden within the photos in such a way that an adversary cannot detect the difference between regular photos and photos with hidden data. Secondly, we need an efficient mapping scheme of the file system blocks to images in order to fully utilize the storage capacity offered by the public server. Finally, covert file access traffic should not be distinguishable from the innocuous photo sharing traffic on the same website, originating from the ordinary users. These users are likely to download new photos and ignore photos they have already seen, they seldom update photos they have already posted and do not delete old photos until there is a shortage of storage on their account. Such access patterns should not be violated when the media website is used to access the hidden file data. In what follows, we will discuss the key design issues that can address these challenges.

2.1 Mounting the File System

In CovertFS, files are stored remotely, hidden within the media hosted on a third party service provider. To access the hidden file system, a user mounts it at a desired mount point in the local file system. Before mounting the file system, the user should have a valid account on a media sharing site. During the mount, the user has to present proper authorization details such as the media sharing website url, account name and password for the account where his file system is hosted and the passphrase for encryption/decryption of the file system contents. After verifying the authorization information, the file system mounts the remote web based file system and begins downloading photos as dictated by the hidden file system accesses. To avoid repeated downloads of certain photos (unusual access pattern for media sharing), photos containing the hidden file system metadata are kept in a local image cache as long as possible.

2.2 Mapping File System Data to Photos

The entire file system information along with file meta data and file contents are encrypted and stored within media content on the service provider. Current steganographic techniques are used to hide the contents of the file within the media. Our media content here are photos to be shared with friends and family. Typical photo sizes stored on Flickr range anywhere between 40KB to 300KB. Current steganographic techniques can safely allow embedding of about 10% of information within a JPEG image with no visual distortion or deviation in statistical properties of the image. Considering a minimum image size of 40 KB, a 4 KB disk block size can be stored. To keep the mapping simple, we can assume that disk blocks are mapped to images one-to-one, which means that we will only hide 4 KB of data even in larger images.

To store all the files within the file system remotely, we need as many images in the Flickr account as the number of blocks on the file system. The number of files that can be stored is constrained by the account storage size within Flickr. However, Flickr and most service providers have unlimited accounts for a minimal service fee per year, providing a virtually unlimited storage capacity. Alternative designs can store mutiple file block in larger images or can span over multiple user accounts and/or multiple service providers.

Metadata, such as inode blocks, and the direct and indirect disk blocks are also stored in photos. Inodes and file block addresses can be identified directly by the name of the image where they are stored or indirectly using inode and block allocation maps, themselves stored in one or multiple images. Retrieval of the photo containing the first block of the map is done through a name that, when hashed, maps to a special value, usually a function of the encryption passphrase entered by the user.

Fig. 1 shows the file system object hierarchy as embedded within different photos stored in the Flickr account. The photo *mountain.jpg* contains the root inode, which points to the only directory inode under the root directory embedded in the photo *hills.jpg*. The direc-

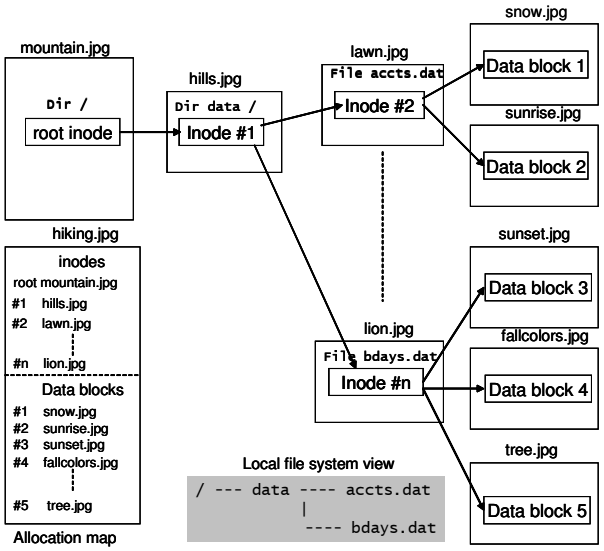


Figure 1: Covert file system layout embedded within Flickr photos and a local view of files when file system is mounted

tory contains two files, whose inodes are embedded in photos *lawn.jpg* and *lion.jpg* respectively. Data blocks are contained within photos *snow.jpg*, *sunrise.jpg*, *sunset.jpg*, *fallcolors.jpg* and *tree.jpg*. The allocation map for inodes and data blocks are stored within the photo *hiking.jpg*. The figure also shows the local view of the file system, within the gray box.

2.3 Handling File System Writes

In a read-write file system, metadata as well as data blocks change as a result of file accesses. In CovertFS, these changes may generate operations that may look suspicious for genuine photo sharing such as (i) frequent image changing and (ii) frequent access to certain old images. In the next two subsections, we will discuss mapping solutions to hide these two file system patterns.

To address frequent image changing due to inode and file system block updates, we propose to make photos immutable and apply an update scheme similar to one used in the log structured file system [10]. According to this scheme, modified file system objects will be hidden in new photos. To achieve this, the indirection through the allocation map is absolutely necessary.

With the proposed scheme, the allocation map becomes the file system object whose frequent changes must also be hidden. To keep the photos carrying the allocation map also immutable, we must devise a mechanism to locate the most recent copy of the map. For this, we propose two complementary schemes. The basic scheme takes advantage of the user-defined name space for photos to apriori decide the name of the photo to store the next version of the map and to embed it along with the version number in the photo of the current map (forward pointer). In this way, a file system user can easily determine when the allocation map has changed by looking at the photo name of the next map. If the new photo does not exist but the old one does, the client can assume that the map has not changed (photos in the same chain are garbage collected in the FIFO order) and use its cached copy. As a backup, in case this chain cannot be reconstructed due to garbage photo collection, the names of the map photos are chosen such that all map to the same special value when when hashed with the user passphrase. In this way, in the worst case, a complete inspection of all the images in the account, will allow a user to discover the most recent copy of the map.

Photo garbage collection is done when the user account reaches near full capacity. The photos containing the invalidated blocks will all be deleted in a batch during this process, freeing up space in the account, yet generating traffic patterns of photo sharing users.

2.4 Avoiding Photo Hotspots

The current design may expose suspicious hotspot patterns as metadata photos are likely to be more frequently accessed, which can be an indicative for a covert file system. Local caching can alleviate this behavior but only partially. To further diffuse this pattern, we plan to introduce forward pointers to all metadata objects and not just the maps. This means that subsequent copies of an inode, for instance, will be chained by embedding the name of the photo to store the next version of the inode in the one carrying the current one. A user who wants to retrieve the most recent version of an inode and has a cached photo of an potentially old version can follow this chain to retrieve it without referring to the allocation map every time. To guarantee that the file corresponding to that inode was not deleted, the most recent copy of the parent directory must also be checked. Finally, avoiding hotspots through this mechanism is an optimization. In case an inode version chain cannot be reconstructed, the user can go back to retrieve the most recent version of the inode starting from the allocation map.

2.5 File Sharing and Access Control

Flickr provides three types of sharing. Photos can either be made private, shared with a group, or made public. Private photos are only accessible to the user who created them. If photos are shared as a group, friends and family can access them and of course, photos made public can be accessed by anybody. However, group and public access sharing do not allow the user to modify the files.

We build our file sharing and access control model on

top of the Flickr photo sharing model. Only the owner of the Flickr account is able to modify file system content, while members of the group or others can only read files or part of the file system that is enabled selectively for read sharing by the owner.

Selective sharing needs to be enabled by the owner who wants to share his files or directories with other users in the group. Each share is assigned a separate encryption passphrase as shown in Fig. 2. The directory Politics is shared with a group of friends with a separate encryption key. Every parent inode object that has a link pointing to a file or directory has a respective encryption key associated with it. Storing the encryption key in the inode allows the owner to access all the files at any time without retyping separate encryption passwords assigned to different shares. In case a separate encryption key is not assigned to any file or directory, the encryption key is replicated from the parent inode. All other directories in Fig. 2 are encrypted with the owners encryption key.

The photos corresponding to the directory to be shared (Politics directory in the fig) are moved to the appropriate category of photos in the Flickr account for sharing with the group. The encryption passphrase for files within the share is given to other users of the group. They can locate the root inode within the share by hashing with the given passphrase. Note that the passphrase is different for each share and can be changed by the owner at any given time, when he decides to revoke sharing.

2.6 Replication

Since the web based services can be unavailable at certain time periods, replicating the file system meta-data and data across different service providers is a desirable design choice. The replicas can be assigned priorities such that the downloads always take place from the primary replica. When the primary replica service provider is unavailable, the files can still be accessed from the secondary replicas. Updates may however be propagated first to the primary and then to the secondary replicas.

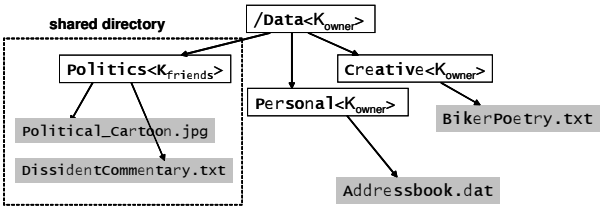


Figure 2: File sharing with CovertFS. The directory *Politics* is shared and has its own encryption key

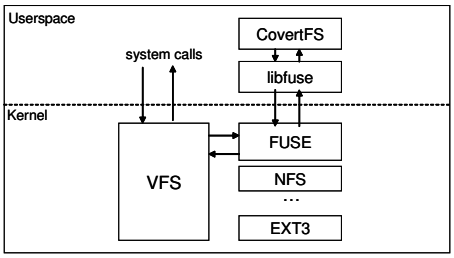


Figure 3: Overview of CovertFS design

2.7 CovertFS and Hidden Levels

It is possible to argue that the presence of CovertFS itself can be incriminating evidence that the user is hiding something. This is countered in the steganographic file systems by having different hidden levels. When compelled by court, the user can disclose only one or two levels with moderately incriminating evidence, while the presence of real data that he wants to hide can be plausibly denied. It is impossible with this design for the examiner to prove that the user is indeed hiding something extra than what he has already disclosed.

A similar analogy applies to CovertFS as well. Hidden levels can be created in CovertFS. Each level has a different encryption passphrase and can only be opened when the user provides the correct passphrase. Additional levels are also mapped by using other photos within the same account. Alternatively, hidden levels may also be created involving additional user accounts on the same or other service providers.

2.8 Implementation Plan

CovertFS can be built as a user-level file system. We plan to implement this on top of the FUSE [1] file system interface as shown in Fig. 3. FUSE facilitates easy development of user-level file systems. It has a kernel mode driver and a user-level library. The user-level library *libfuse* interacts with the kernel mode driver through a device called */dev/fuse*. The system calls that operate on files in the FUSE file system are redirected from the virtual file system (VFS) layer in the kernel to the FUSE driver. The driver in turn forwards this call to the user-space library. The new filesystem, CovertFS in the figure, that links into this library can handle this call and implement new functionality. We plan to develop our proof-of-concept prototype over Flickr [2], since they have a public API available for this purpose.

3 Discussion

In this section, we provide security analysis and discuss other design related issues.

3.1 Security Analysis

We define two types of adversaries. A passive adversary simply observes the traffic and checks for anomalies. An active adversary, on the other hand, actively performs steganalysis on random images from time to time to detect hidden data within the images. We examine why CovertFS is indeed covert from the point of view of both the active and the passive adversary. The active adversary is primarily concerned with steganalysis, while the passive adversary mainly performs traffic analysis on the Flickr account traffic.

3.1.1 Active Adversary and Steganalysis

Steganalysis is a technique where the adversary can determine that the image is used as a cover for hidden information. Steganalysis techniques watch for signature distortions created by known steganography tools. With respect to JPEG images, some steganography tools use simple techniques such as LSB encoding. In this method each bit of the hidden text is encoded in the least significant bit of every byte in the JPEG image. This technique does not cause visual distortion perceptive to the human eye, but creates huge deviations in the statistical properties of the JPEG image. Such tools that perform bit manipulations are called image domain tools and can be detected easily by steganalysis.

Other set of techniques used for hiding information is called the transform domain tools. These group of tools use techniques that involve manipulation of algorithms and image transforms. One of the popular techniques used for JPEG images is called the discrete cosine transform (DCT). These methods hide information in more significant areas of the image and may manipulate image properties such as luminance. These techniques are far more robust and much harder to detect using steganalysis. The tradeoff however is that such methods can encode much smaller amount of information within the cover. Recent research has come up with zero divergence steganography or secure steganography that use statistical restoration techniques [13, 12, 11]. The basic idea is to thwart steganalysis methods by hiding information in few bits within the image and adjusting other bits to offset the deviations caused by the hidden information. Hence, advances in steganography have made it possible to build tools that can thwart steganalysis. We plan to use one such advanced technique in our prototype to hide the file system data.

3.1.2 Passive Adversary and Traffic Analysis

The passive adversary simply sniffs traffic to look for anomalies and tries to deduce if any hidden text exists within the image. All the traffic during upload and download of the files within the file system must appear like innocuous photo sharing traffic. However, the pattern in which the files are accessed may leak some information to the adversary. The adversary however, must not be able to determine with certainty that a specific pattern at the beginning of accesses, implies hidden text.

Traffic patterns can be obfuscated by introducing pseudo random dummy image fetches. The client can cache already visited photos to ensure that it does not download those photos too frequently. CovertFS is designed such that only new photos are uploaded and old ones are deleted when the account reaches near full capacity, which resembles the behavior of normal photo sharing users. Also, the additions are done in a batch as the file system operates in a disconnected mode, making additions in a batch to the photo store, similar to how regular users add photos. Since Flickr has an open API, several other applications have been built on top of it that perform specific tasks, customized to the user. Each of these tasks gives rise to different upload/download patterns.

3.2 Feature or Misuse

CovertFS can be built on top of media sharing service such as Flickr. While this provides an innovative use of a commonly used web service, this can be viewed as abuse of a service designed for a different purpose. We argue that since Flickr is a photo sharing service, what else is embedded in the photos does not really affect Flickr's business model. Users still host photos on Flickr for CovertFS to work.

4 Related Work

The steganographic file system that gives the user plausible deniability was first proposed by Anderson and Shamir [6]. They did not have a working prototype of the file system. McDonald et al [8] were the first to build a working prototype of a steganographic file system called StegFS. StegFS is a local file system that provides plausible deniability by hiding files in unused disk blocks. The prototype did not require a separate partition but worked along with the Linux ext2 partition. Pang et al [9] demonstrated improvements to the hiding schemes and design of StegFS, which demonstrated significant improvements in performance. All the file systems mentioned above work with the local hard drive and provide plausible deniability to the user. None of these provide the ability to globally access or share files. Since all of these hide in unused disk blocks, they run the risk of being overwritten when the driver is not operating in the steganographic mode. Therefore these require a high degree of replication, severely limiting the disk space usage. CovertFS, on the other hand, provides file sharing

between geographically distant users as well as plausible deniability. CovertFS hides files from the service providers themselves and is built over a media sharing service. The design considerations are significantly different in both cases.

The gmail file system [3] allows the user to store his data as email messages in his mail account. The service provider is aware of the existence of the user files in this mail account. This file system does not allow plausible deniability or enable file sharing with others. Httpfs [7] is a network file system that provides access to files on a remote machine using the http protocol. It requires a component to run on the remote server, from where documents can be fetched on the client. This is similar to the network file system implementation but using http. For CovertFS, no such component is required on the server side. DavFS [4] allows to mount files from a WebDAV server on a local driver. WebDAV is an extension of http that allows remote collaborative authoring of web resources. DavFS allows a remote web server to be edited simultaneously by a group using standard applications. DavFS, fundamentally differs from our implementation as it requires a server component. None of the above file systems provide plausible deniability either. CovertFS can run on top of any media hosting service. The control lies with the user on how he accesses/modifies his hidden files.

The Web File system [5] provides a file system interface to the world wide web. The goal here is completely different from our goal. This file system allows the user to browse the web as different files that are downloaded on the local hard drive.

5 Conclusion

In this paper, we motivate the need for a web based covert file system, CovertFS. This file system allows users to store their files, hidden inside the media hosted on a public server and access them from anywhere in the world with complete confidentiality from any third party including the service provider. Additionally, the very existence of the file system is known only to the user and cannot be determined or proven by anyone else. Further, it allows files to be selectively and covertly shared with others as and when needed. As part of ongoing work, we are developing a working prototype of such a file system and will evaluate it in terms of latencies, scalability, security and privacy.

Acknowledgments

We would like to thank the anonymous reviewers and Pandurang Kamat for their insightful comments and feedback. This work has been supported in part by the NSF CAREER grant CCR-0133366.

References

[1] Filesystem in userspace. http://fuse.sourceforge.net/.

[2] Flickr photo sharing. http://www.flickr.com/.

[3] The gmail file system. http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html.

[4] Webdav linux file system (davfs2). http://dav.sourceforge.net/.

[5] ADYA, A. Web file system: File-like access to the web. In *5th Annual MIT Student Workshop on Scalable Computing. Wellesley, MA. August 1995*.

[6] ANDERSON, NEEDHAM, AND SHAMIR. The steganographic file system. In *IWIH: International Workshop on Information Hiding* (1998).

[7] KISELYOV, O. A network file system over HTTP: Remote access and modification of files and files. pp. 75–80.

[8] MCDONALD, A. D., AND KUHN, M. G. Stegfs: A steganographic file system for linux. In *Information Hiding* (1999), pp. 462–477.

[9] PANG, H., TAN, K.-L., AND ZHOU, X. Stegfs: A steganographic file system. *icde 00* (2003), 657.

[10] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1991), ACM Press, pp. 1–15.

[11] SOLANKI, K., SULLIVAN, K., MADHOW, U., MANJUNATH, B. S., AND CHANDRASEKARAN, S. Statistical restoration for robust and secure steganography. In *IEEE International Conference on Image Processing* (Sep 2005).

[12] SOLANKI, K., SULLIVAN, K., MADHOW, U., MANJUNATH, B. S., AND CHANDRASEKARAN, S. Provably secure steganography: Achieving zero k-l divergence using statistical restoration. In *IEEE International Conference on Image Processing 2006 (ICIP06)* (Oct 2006).

[13] SULLIVAN, K., SOLANKI, K., MANJUNATH, B., MADHOW, U., AND CHANDRASEKARAN, S. Determining achievable rates for secure zero divergence steganography. In *IEEE International Conference on Image Processing 2006 (ICIP06)* (Oct 2006).

Purely Functional System Configuration Management

Eelco Dolstra
Utrecht University
http://www.cs.uu.nl/~eelco

Armijn Hemel
Loohuis Consulting
armijn@loohuis-consulting.nl

Abstract

System configuration management is difficult because systems evolve in an undisciplined way: packages are upgraded, configuration files are edited, and so on. The management of existing operating systems is strongly *imperative* in nature, since software packages and configuration data (e.g., /bin and /etc in Unix) can be seen as imperative data structures: they are updated in-place by system administration actions. In this paper we present an alternative approach to system configuration management: a *purely functional* method, analogous to languages like Haskell. In this approach, the static parts of a configuration — software packages, configuration files, control scripts — are built from pure functions, i.e., the results depend solely on the specified inputs of the function and are immutable. As a result, realising a system configuration becomes deterministic and reproducible. Upgrading to a new configuration is mostly atomic and doesn't overwrite anything of the old configuration, thus enabling rollbacks. We have implemented the purely functional model in a small but realistic Linux-based operating system distribution called NixOS.

1 Introduction

A system configuration is the composition of artifacts necessary to make computer systems performs their intended functions. Managing the configuration of a system is difficult because there are typically thousands of these interrelated artifacts that together make up a system (such as software packages, configuration data, and control scripts), and we need to manage the evolution of such a configuration.

Figure 1 shows a very small subset of the various kinds of artifacts that make up a simple Linux system running an OpenSSH server and Apache (providing access to Subversion repositories). Boxes denote software components (including scripts), solid ellipses denote configuration files, and dashed ellipses are conceptual groupings of configuration aspects. An arrow $a \rightarrow b$ indicates that

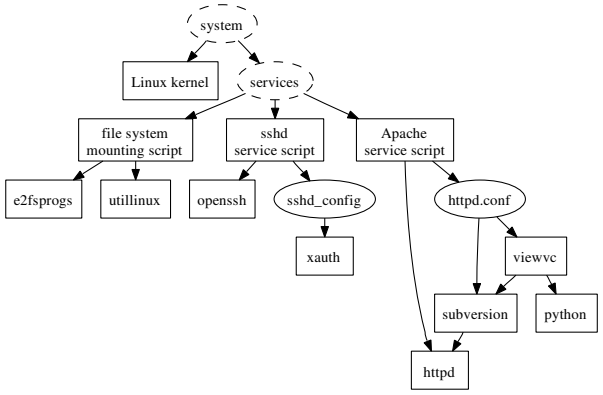


Figure 1: A small subset of a system configuration

system component *a* has some kind of reference to component *b*, be it a dynamic linker dependency, a line in a configuration file, and so on.

In this paper we argue that existing configuration management tools — ranging from package managers such as RPM [7] to configuration tools such as Cfengine [2] — have an *imperative model* in a sense analogous to imperative programming languages such as C. That is, configuration actions such as upgrading a package or modifying a configuration file are *stateful*: they depend on and transform the state of the system.

Statefulness has a number of serious consequences:

- No *traceability*: if a configuration is the result of a sequence of imperative actions over time (some of which may have been performed manually), then there may not be a record of these actions that allows the configuration to be recreated on an empty machine. As a result, it is hard to *reproduce* configurations.

- No *predictability*: if a configuration action depends on an ill-defined initial state, then the end result may be unpredictable. This is why upgrading an operating system is so much more error-prone than reinstalling it: the upgrade action transforms a poorly defined initial state, the result often being equally poorly defined,

while the installation action starts from a well-defined initial state.

- Inability to run multiple configurations *side-by-side*. For instance, if we want to test some modification to the Apache server in Figure 1, then we could of course modify the httpd.conf of the production instance to run on a different port. But this would affect the production instance as well. Instead, we should copy httpd.conf, but this propagates through the dependency graph; for instance, we also would have to copy the Apache service script to refer to the new httpd.conf.

- A special instance of the previous problem is the inability to *roll back* the system to a previous configuration. For instance, if we upgrade a set of RPM packages and the upgrade turns out to be less than desirable, then undoing the change is hard: we would have to revert to backups (but it may not be clear which files to restore) or install the previous RPM packages (if we know which ones!). Likewise, if we make a bad modification to a configuration file, we'd better have the file under version control to revert the change.

Analogous problems exist in imperative programming languages, such as the inability to reason about the result of function calls due to global variables or I/O. This was an important motivation for the development of purely functional programming languages [11] like Haskell [12]. In those languages, the result of a function call only depends on the inputs of the function, and values are immutable. This makes it easier to reason about the behaviour of a program. For instance, two function calls $f(x)$ and $f(y)$ can never interfere with each other (e.g., because of mutable global variables or I/O), and $x = y$ implies $f(x) = f(y)$. This property is known as *referential transparency* [11], and it is what is lacking in conventional system configuration tools. For instance, when a configuration file such as sshd_config has a reference to some path, say /usr/X11/bin/xauth, then the *referent* (the file being pointed to) is not constant. Thus a configuration action in one corner of the system (such as uninstalling xauth) can affect the behaviour in another. This is what causes problems such as the "DLL hell".

In this paper, we show that it is possible to do system configuration management in a purely functional way. This means that the static parts of a configuration (software packages, configuration files, control scripts) are built from pure functions and *never change after they have been built*. This has a number of advantages:

- A configuration can be realised (*built*) deterministically from a single formal description. This also means that a configuration can be reproduced easily on another machine.

- Upgrading a configuration is as safe as installing from scratch, since the realisation of a configuration is not stateful.

- Configuration changes are not destructive. As a result, we can always roll back to a previous configuration that has not been garbage collected yet.

To show that a purely functional model is feasible, we have implemented a small but realistic Linux-based operating system distribution called *NixOS*, built on the purely functional package management system *Nix*. In NixOS, all static parts of the system are stored as immutable "values" under paths such as

```
/nix/store/2m732xrk...-apache-2.2.3
```

where 2m732xrk... is a cryptographic hash of the inputs involved in building the value. Aside from a single exception, there is no /bin, /usr, /lib, etc. in this system, and /etc consists almost entirely of symlinks to generated configuration files in /nix/store. The remainder of this paper shows the basic principles behind NixOS.

2 Purely functional package management

NixOS is based on Nix, a purely functional package management system [5, 6]. *Nix expressions* describe how to build immutable software packages from source. For instance, the following Nix expression is a *function* that builds Apache:

```
{stdenv, openssl}:

stdenv.mkDerivation {
  name = "apache-2.2.3";
  src = fetchurl {
    url = http://.../httpd-2.2.3.tar.bz2;
    md5 = "887bf4a85505e97b...";
  };
  buildCommand = "
    tar xjf $src
    ./configure --prefix=$out \
      --with-openssl=${openssl}
    make; make install";
}
```

Here, stdenv and openssl are function arguments representing dependencies of Apache (stdenv is a standard build environment: GCC, Make, etc.). When the function is called, it builds a *derivation*, which is an atomic build action. In this case, the build action unpacks, configures, compiles and installs Apache. All attributes

specified in the derivation (such as src) are passed to the builder through environment variables. The out environment variable contains the target path of the package, discussed below. Dependencies such as src, stdenv and openssl are recursively built before Apache's derivation is built.

To build a concrete Apache instance, we write an expression that calls the function:

```
apache = import ./apache.nix {
  inherit stdenv openssl;
}
stdenv = ...;
openssl = ...;
```

That is, the file containing the Apache function is imported and called with specific instances of stdenv and openssl, which are defined similarly. (inherit simply copies the values of stdenv and openssl from the surrounding lexical scope. The details of the language aren't very important here; the interested reader is referred to [5].) The user can now do

```
$ nix-env -f all-packages.nix -i apache
```

to install the Apache package.

The Nix expression language happens to be purely functional, but what really matters is that *the storage of packages is also purely functional*. Packages are built in the *Nix store*, a designated part of the file system, typically /nix/store. Each package is stored separately under a name that contains a 160-bit cryptographic hash of the inputs involved in building the package, e.g., /nix/store/5lbfaxb7...-openssl-0.9.8d or /nix/store/2m732xrk...-apache-2.2.3. This location is passed to the build script through the out environment variable. As a result, any change to any input causes a different path, so if we build the Nix expression, the package will be rebuilt in a different path in the store. Any previous installations of the package are left untouched, and so we prevent problems like the "DLL hell". On the other hand, if the path already exists, then we can safely skip rebuilding it.

Packages are made read-only after they have been built. This is why the Nix store can be called purely functional: the cryptographic hash in the path of a package is defined by the inputs to the package's build process, and the contents never change; thus there is a unique correspondence between the hash and the contents. This scheme has several important advantages [6], such as preventing undeclared build time dependencies, allowing detection of runtime dependencies, and allowing automatic garbage collection of unused packages.

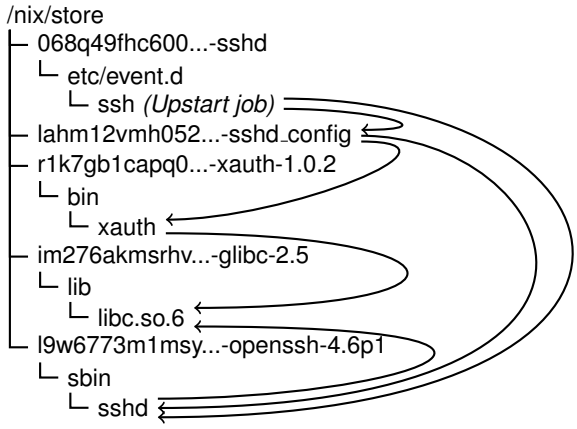


Figure 2: Paths in the Nix store related to the SSH service

3 System configuration management

We can quite naturally extend purely functional package management to purely functional system configuration management, simply by treating the other parts of a configuration — such as configuration files and control scripts — as packages.

For instance, consider the configuration file for the sshd daemon, sshd_config. We can make a trivial Nix expression that builds it in the Nix store:

```
{stdenv, xauth}:
stdenv.mkDerivation {
  name = "sshd_config";
  buildCommand = "
    echo 'X11Forwarding yes' > $out
    echo 'XAuthLocation \
      ${xauth}/bin/xauth ' >> $out";
}
```

(The construct ${xauth} places the store path of the xauth argument in the enclosing string. The actual implementation in NixOS uses more sophisticated configuration file templating mechanisms.)

In the same way that we built sshd_config purely, we can build all the other static parts that constitute a system, such as the configuration in Figure 1. Thus there are Nix expressions to build the kernel, the initial ramdisk (initrd) necessary for providing boot-time modules required by the kernel to mount the root file system, the boot scripts, the Upstart jobs[1], the X server plus its configuration, etc.

For example, the SSH service in Figure 1 is realised through Nix expressions that build an Upstart

[1] Upstart is a event-based replacement for the classic /sbin/init (http://upstart.ubuntu.com/). It's responsible for running system startup scripts and monitoring system daemons.

job that starts the SSH daemon, the SSH configuration file, the OpenSSH and Xauth packages, and their software dependencies such as the C library Glibc. The results in the Nix store of building these expressions are shown in Figure 2 (most software dependencies are omitted). The arrows denote runtime dependencies that arise from a value containing the path to another value in the store. For instance, the generated Upstart job contains a line /nix/store/l9w6773m1msy...-openssh-4.6p1/sbin/sshd -f /nix/store/lahm12vmh052...-sshd_config, and so it has a runtime dependency on OpenSSH and the configuration file.

There is also a top-level Nix expression, system.nix, that builds the entire system configuration by calling the individual expressions that build specific parts of the system. The output of this expression is a package containing an *activation script* that makes the configuration the current configuration of the system. For instance, it modifies the Grub startup menu so that the system will boot with the new configuration the next time the system is booted. The Grub menu also contains all previous configurations that have not been garbage collected yet, allowing the user to go back to old configurations very easily if there is a problem with the new configuration.

Of course, most configuration changes do not require the system to be restarted. A nice property of NixOS's purely functional model is that it allows the activation script to determine precisely which system services have to be restarted, simply by comparing the store paths of their Upstart job files. After all, due to the immutability of files in the Nix store, if an Upstart job with a certain name (e.g., sshd) has the same path in the new configuration as in the previous configuration, then it must be the same.

When the user changes anything to a Nix expression, she can realise the new configuration as follows:

```
$ nixos-rebuild switch
```

This builds system.nix, makes it the default configuration for booting, and calls the resulting activation script. Similarly, nixos-rebuild test builds and activates a configuration but does not make it the boot default. Thus, a reboot suffices to recover from a crashing configuration. Also, since the building of a new configuration is non-destructive — it does not overwrite any existing files in the store — the user can roll back to any previous configuration (that has not been garbage collected yet) by running its activation script.

An advantage of using a functional language is that it is easy to abstract over configuration choices. Rather than encoding those choices directly in a Nix expression, they can be passed as function arguments to the expression. For instance, whether to turn on X11 forwarding for sshd can be passed as a function argument:

```
{stdenv, xauth, forwardX11}:
stdenv.mkDerivation {
  name = "sshd_config";
  buildCommand = "
    ${if forwardX11 then
        "echo 'X11Forwarding yes' > $out
         echo 'XAuthLocation \
           ${xauth}/bin/xauth ' >> $out"
      else ""}";
}
```

This causes the line echo 'XAuthLocation ...' to be included in the build command only if forwardX11 is true. This has the additional advantage that due to lazy evaluation (derivations are only built when they are actually referenced), xauth is built only when forwardX11 is set. This kind of optimisation follows from the integration of package management and system configuration management into a single formalism.

In the same vein, system.nix is a function that accepts a *configuration specification*, which is a hierarchical set of attributes specifying various system options. It passes these options on to the appropriate Nix expressions, e.g.,

```
{
  boot = {
    grubDevice = "/dev/hda";
  };
  fileSystems = [
    { mountPoint = "/";
      device = "/dev/hda1";
    }
  ];
  swapDevices = ["/dev/hdb1"];
  services = {
    sshd = {
      enable = true;
      forwardX11 = true;
    };
    apache = {
      enable = true;
      subservices = {
        subversion = {
          enable = true;
          dataDir = "/data/subversion";
        };
      };
    };
  };
}
```

Figure 3: A simple configuration specification

services.sshd.forwardX11 is passed on to the function that builds the sshd service. Figure 3 shows a real example of a configuration specification corresponding to the configuration in Figure 1.

An important advantage of our approach is that since the entire system configuration is expressed in a single formalism, a user does not have to know how a specific configuration choice is implemented. Regardless of the configuration change that the user wants to perform — upgrading to a new version of OpenSSH, changing an sshd configuration option, upgrading to a new kernel, adding or removing a file system or swap device — the change is accomplished in the same way: one modifies the configuration file (Figure 3) and reruns nixos-rebuild to build and activate the configuration.

4 Evaluation

Does the purely functional approach work? That is, to what extent can we eliminate the "global" namespace of files in /etc, /bin and so on and replace them with purely built, immutable files?

Quite well, in fact. NixOS is currently a somewhat small but realistic Linux distribution. It builds on the Nix Packages collection which contains some 850 Unix packages. NixOS provides a Linux 2.6-based system, networking, system services such SSH and Apache, an X server, KDE and parts of Gnome. It works well enough that it has replaced SUSE on the first author's laptop. It is also in use as a server OS on the *build farms* (automated software build systems) at Utrecht University and the Technical University of Delft. (Nix is useful on such systems as it manages the deployment of dependencies to build machines and prevents undeclared dependencies in software packages.)

Code With regard to software, NixOS has no /usr, /sbin, or /lib. There is only one file in /bin: namely, a symlink /bin/sh to a Bash instance in the Nix store. This is because many programs (such as Glibc's system() function) hard-code the location of the shell. Of course, we could patch all those programs, but instead we took the pragmatic route — a slight concession to the purely functional model. Apart from the symlink /bin/sh, all packages resides in the Nix store, and have no dependencies on packages outside of the store. For example, once we had NixOS bootstrapped with just the single /bin/sh compromise, we were able to build Mozilla Firefox and all its dependencies all the way to Glibc and GCC purely. (Details of the NixOS bootstrap can be found in [9].)

Static data How about configuration data in /etc? A lot of configuration data can be "purified" easily. For instance, a configuration file such as /etc/ssh/sshd_config can easily be generated in a Nix expression and used directly from the Nix store by also generating an Upstart job that calls sshd with the appropriate -f argument to specify the Nix store path of the generated sshd_config.

However, there are some configuration files for which this is either not possible (e.g., because the path is hard-coded into binaries) or infeasible (because the configuration cross-cuts the system). Examples are /etc/services (well-known port numbers) and /etc/resolv.conf (DNS configuration). We *do* generate those files in Nix expressions, but the activation script of the configuration creates symlinks to them in /etc. The X11/KDE-based configuration of the first author's laptop has 24 symlinked files and directories in /etc, a number that can probably be reduced with little effort. This configuration, when built, consists of 236 paths in the Nix store; its build-time dependency graph consists of 532 build actions.

Mutable state Since files in the Nix store are immutable, the purely functional approach only works for the static parts of a configuration. Mutable state, such as most everything in /var, falls outside the scope of this approach. So mutable state isn't modeled in Nix expressions explicitly; rather, programs such as the activation script, Upstart jobs or system daemons are responsible for initialising the required state in /var at runtime.

Some files occupy a place between static configuration and mutable state. A prominent example is the Unix user account database, /etc/passwd. On the one hand, it specifies static configuration such as the existence of system accounts; on the other hand, passwords are dynamically modified at runtime by end users through commands such as passwd. Currently, the activation script ensures that the required accounts exist, but this is a stateful operation that depends on the previous contents of /etc/passwd and therefore has all the problems associated with stateful updating. Fortunately, there are only 3 such files in the example configuration above.

Disk space The purely functional model has significant disk space requirements. For instance, the 236 store paths of the X11/KDE-based configuration above have a total size of 656 MiB. This is not remarkable in itself, given that the configuration contains a kernel, an X server, KDE, and many other packages. However, if we make a change to this configuration, then in the worst case — if every derivation is different — we need another 656 MiB of disk space. In an imperative model, the new configuration simply overwrites the old configuration, and so no additional space is needed.

Fortunately, most configuration changes do not need anywhere near that amount of space, as they only cause

"top level" configuration files and control scripts to be rebuilt, taking up only a few kilobytes. However, the worst case can occur, for instance if the C library or compiler is changed (which are used by all other packages). This is analogous to efficiency considerations for purely functional data structures: it is inefficient to update the last element of a Haskell list, but cheap to update the first element. Closures in the Nix store are purely functional data structures, too, so it is cheap to update something near the top of the dependency graph (e.g., configuration files) but expensive to update things near the bottom (e.g., the C library).

5 Related Work

Our work on NixOS is an extension of our previous work on purely functional software deployment [6, 5]. We extended software deployment to service deployment in [4]. The latter paper briefly discusses deploying distributed applications with Nix.

The Vesta configuration management system [10] has a purely functional language for build management. It should be possible to use this language to build system configurations.

The need to make system configuration more declarative has been felt by many. An approach inspired by Vesta is suggested in [3]. The present work could be seen as a realisation of that idea. Configuration tools such as Cfengine [2] and LCFG [1] are declarative, but stateful. For instance, Cfengine actions transform the current state of the system.

While to our knowledge this is the first attempt at purely functional system configuration management, there have been operating systems written *in* a functional language, such as House [8]. This is a rather orthogonal issue; the system configuration of those OSes is managed in a conventional way.

6 Conclusion

We have shown that it is possible to implement an operating system with a purely functional configuration management model, where all software packages and configuration files are built from a formal description of the system using pure functions and are never changed after they have been built. Hudak [11] points out that people unfamiliar with functional languages often find it hard to believe that it is possible to live without assignments; likewise, it may not be entirely obvious that one can manage an operating system without destructively updating files. We have shown that this is in fact quite possible. Initial experience with NixOS suggests that the approach is also practical.

NixOS, including sources and ISO images for 32-bit and 64-bit x86 machines, is available from http://nix.cs.uu.nl/nixos.

Acknowledgements This research was supported by NWO/JACQUARD project 638.001.201, *TraCE: Transparent Configuration Environments*. We wish to thank Martin Bravenboer, Eelco Visser, Rob Vermaas and others for contributing to the development of Nix, and the anonymous reviewers for their helpful comments.

References

[1] P. Anderson and A. Scobie. LCFG: The next generation. In *UKUUG Winter Conference*, Feb. 2002.

[2] M. Burgess. Cfengine: a site configuration engine. *Computing Systems*, 8(3), 1995.

[3] J. DeTreville. Making system configuration more declarative. In *HotOS X, Tenth Workshop on Hot Topics in Operating Systems*. USENIX, June 2005.

[4] E. Dolstra, M. Bravenboer, and E. Visser. Service configuration management. In *12th Intl. Workshop on Software Configuration Management (SCM-12)*, Sept. 2005.

[5] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In L. Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, Nov. 2004. USENIX.

[6] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *26th Intl. Conf. on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.

[7] E. Foster-Johnson. *Red Hat RPM Guide*. John Wiley & Sons, 2003. Also at http://fedora.redhat.com/docs/drafts/rpm-guide-en/.

[8] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in Haskell. In *ICFP '05: Tenth ACM SIGPLAN Intl. Conf. on Functional Programming*, pages 116–128. ACM Press, 2005.

[9] A. Hemel. NixOS: the Nix based operating system. Master's thesis, Dept. of Information and Computing Sciences, Utrecht University, Aug. 2006. INF/SCR-2005-091.

[10] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *ACM SIGPLAN '00 Conf. on Programming Language Design and Implementation*, pages 311–320. ACM Press, 2000.

[11] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[12] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Apr. 2004.

Processor Hardware Counter Statistics As A First-Class System Resource *

Xiao Zhang Sandhya Dwarkadas Girts Folkmanis Kai Shen
Department of Computer Science, University of Rochester

Abstract

Today's processors provide a rich source of statistical information on program execution characteristics through hardware counters. However, traditionally, operating system (OS) support for and utilization of the hardware counter statistics has been limited and ad hoc. In this paper, we make the case for direct OS management of hardware counter statistics. First, we show the utility of processor counter statistics in CPU scheduling (for improved performance and fairness) and in online workload modeling, both of which require online continuous statistics (as opposed to ad hoc infrequent uses). Second, we show that simultaneous system and user use of hardware counters is possible via time-division multiplexing. Finally, we highlight potential counter misuses to indicate that the OS should address potential security issues in utilizing processor counter statistics.

1 Introduction

Hardware counters are commonplace on modern processors, providing detailed information such as instruction mix, rate of execution (instructions per cycle), branch (control flow) prediction accuracy, and memory access behaviors (including miss rates at each level of the memory hierarchy as well as bus activity). These counters were originally provided for hardware verification and debugging purposes. Recently, they have also been used to support a variety of tasks concerning software systems and applications, including adaptive CPU scheduling [2, 6, 11, 15, 18], performance monitoring/debugging [1, 5, 19], workload pattern identification [4, 7], and adaptive application self-management [8].

Except for guiding CPU scheduling, so far the operating system's involvement in managing the processor counter statistics has been limited. Typically the OS does little more than expose the statistics to user applications. Additional efforts mostly concern the presentation of counter statistics. For instance, the PAPI project [5] proposed a portable cross-platform interface that applications could use to access hardware events of interests, which hides the differences and details of each hardware platform from the user.

In this paper, we argue that processor hardware counters are a first-class resource, warranting general OS utilization and requiring direct OS management. Our discussion is within the context of the increasing ubiquity and variety of hardware resource-sharing multiprocessors. Examples are memory bus-sharing symmetric multiprocessors (SMPs), L2 cache-sharing chip multiprocessors (CMPs), and simultaneous multithreading (SMTs), where many hardware resources including even the processor counter registers are shared.

Processor metrics can identify hardware resource contention on resource-sharing multiprocessors in addition to providing useful information on application execution behavior. We reinforce existing results to demonstrate multiple uses of counter statistics in an online continuous fashion. We show (via modification of the Linux scheduler) that on-line processor hardware metrics-based simple heuristics may improve both the performance and the fairness of CPU scheduling. We also demonstrate the effectiveness of using hardware metrics for application-level online workload modeling.

A processor usually has a limited number of counter registers to which a much larger number of hardware metrics can map. Different uses such as system-level functions (*e.g.*, CPU scheduling) and user-level tasks (*e.g.*, workload profiling) may desire conflicting sets of processor counter statistics at the same time. We demonstrate that such simultaneous use is possible via time-division multiplexing.

Finally, the utilization of processor counter statistics may bring security risks. For instance, a non-privileged user application may learn execution characteristics of other applications when processor counters report combined hardware metrics of two resource-sharing sibling processors. We argue that the OS should be aware of such risks and minimize them when needed.

2 Counter Statistics Usage Case Studies

We present two usage case studies of processor hardware counter statistics: operating system CPU scheduling and online workload modeling. In both cases, the

*This work was supported in part by NSF grants CNS-0411127, CCF-0448413, CNS-0509270, CNS-0615045, CNS-0615139, and CCF-0621472.

processor counter statistics are utilized in a continuous online fashion (as opposed to ad hoc infrequent uses).

2.1 Efficient and Fair CPU Scheduling

It is well known that different pairings of tasks on resource-sharing multiprocessors may result in different levels of resource contention and thus differences in performance. Resource contention also affects fairness since a task may make less progress under higher resource contention (given the same amount of CPU time). A fair scheduler should therefore go beyond allocating equal CPU time to tasks. A number of previous studies [2, 6, 10, 11, 15, 18] have explored adaptive CPU scheduling to improve system performance and some have utilized processor hardware statistics. The case for utilizing processor counter statistics in general CPU scheduling can be strengthened if a counter-based simple heuristic improves both scheduling performance and fairness.

In this case study, we consider two simple scheduling policies using hardware counter statistics. The first (proposed by Fedorova *et al.* [11]) uses instruction-per-cycle (IPC) as an indicator of whether a task is CPU-intensive (high IPC) or memory-access-intensive (low IPC). The IPC scheduler tries to pair high-IPC tasks with low-IPC tasks to reduce resource contention. The second is a new policy that directly measures the usage on bottleneck resources and then matches each high resource-usage task with a low resource-usage task on resource-sharing sibling processors. In the simple case of SMPs, a single resource — the memory bus — is the bottleneck.

Our implementation, based on the Linux 2.6.10 kernel, requires only a small change to the existing CPU scheduler. We monitor the bus utilization (or IPC) of each task using hardware counter statistics. During each context switch, we try to choose one ready task whose monitored bus utilization (IPC) is complementary to the task or tasks currently running on the other CPU or CPUs (we use last-value prediction as a simple yet reasonable predictor, although other more sophisticated prediction schemes [9] could easily be incorporated). Note that our implementation does not change the underlying Linux scheduler's assignation of equal CPU time to CPU-bound tasks within each scheduling epoch. By smoothing out overall resource utilization over time, however, the scheduler may improve *both* fairness *and* performance, since with lower variation in resource contention, tasks tend to make more deterministic progress.

Experimental results We present results on CPU scheduling in terms of both performance and fairness using two sets of workloads — sequential applications (SPEC-CPU2000) and server applications (a web server workload and TPC-H). The test is performed on an SMP system consisting of 2 Intel Xeon 3.0 GHz CPUs with Hyper-Threading disabled.

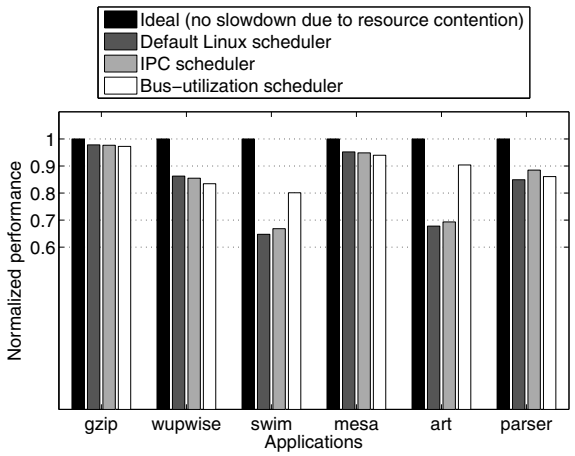


Figure 1: Normalized performance of individual SPEC-CPU2000 applications under different scheduling schemes.

For experiments on SPEC-CPU2000 applications, we run gzip, parser, and swim (low, medium, and high bus utilization, respectively) on one CPU, and mesa, wupwise, and art (again, low, medium, and high bus utilization, respectively) on the other CPU. In this scenario, ideally, complementary tasks (high-low, medium-medium) should be executed simultaneously in order to smooth out resource demand. We define the normalized application performance as "$\frac{\text{execution time under ideal condition}}{\text{execution time under current condition}}$". The ideal execution time is that achieved when the application runs alone (with no processor hardware resource contention). Figure 1 shows the normalized performance of SPEC-CPU2000 applications under different schedulers.

We define two metrics to quantify the overall system performance and fairness. The *system normalized performance* metric is defined as the geometric mean of each application's normalized performance. The *unfairness factor* metric is defined as the coefficient of variation (standard deviation divided by the mean) of all application performance. Ideally, the system normalized performance should be 1 (*i.e.*, no slowdown due to resource contention) and the unfairness factor 0 (*i.e.*, all applications are affected by exactly the same amount). Compared to the default Linux scheduler, the bus-utilization scheduler improves system performance by 7.9% (from 0.818 to 0.883) and reduces unfairness by 58% (from 0.178 to 0.074). Compared to the IPC scheduler, it improves system performance by 6.5% and reduces unfairness by 55%. The IPC scheduler is inferior to the bus-utilization scheduler because IPC does not always accurately reflect the utilization of the shared bus resource.

We also experimented with counter statistics-assisted CPU scheduling using two server applications. The first

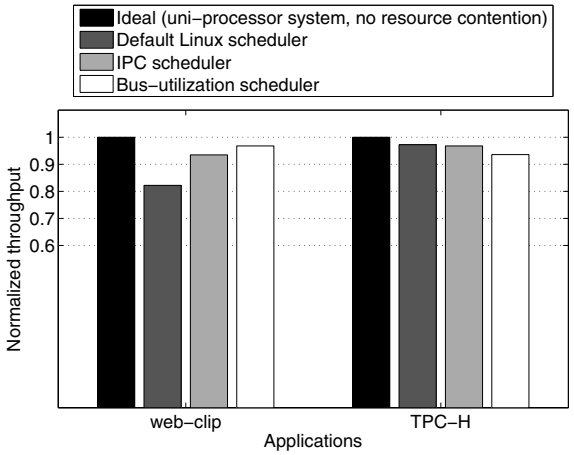


Figure 2: Normalized throughput of two server applications under different scheduling schemes.

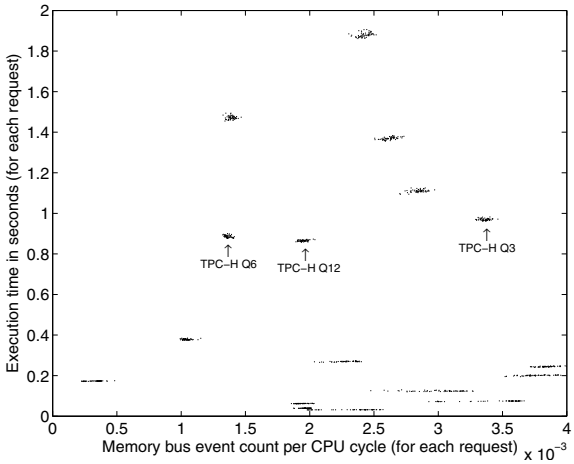


Figure 3: TPC-H request modeling using execution time and memory access intensity (measured by the Intel Xeon processor counter metric FSB_DATAREADY per CPU cycle).

is the Apache 2.0.44 web server hosting a set of video clips, synthetically generated following the file size and access popularity distribution of the 1998 World Cup workload [3]. We call this workload *web-clip*. The second application is the TPC-H benchmark running on the MySQL 5.0.17 database. We choose a subset of 17 relatively short TPC-H queries appropriate for an interactive server workload. The datasets we generated for the two workloads are at 316 MB and 362 MB respectively. For our experiments, we first warmup the server memory so that no disk I/O is performed during performance measurement. Figure 2 presents server throughput normalized to that when running alone without resource contention (ideal). Compared to the default Linux scheduler, the bus-utilization scheduler improves system throughput by 6.4% (from 0.894 to 0.952) and reduces unfairness by 80% (from 0.118 to 0.024). In this case, the IPC scheduler's performance is close to that of the bus-utilization scheduler, with system throughput at 0.951 and an unfairness factor of 0.025.

2.2 Online Workload Modeling

In a server system, online continuous collection of per-request information can help construct workload models, classify workload patterns, and support performance projections (as shown in Magpie [4]). Past work has mostly focused on the collection of software metrics like request execution time. We argue that better server request modeling may be attained by adding processor hardware counter statistics. We support this argument using a simple experiment. We ran our TPC-H workload for 10 minutes and monitored around 1000 requests. For each request, we collected its execution time and memory bus utilization (measured using processor hardware counters). The per-request plot of these two metrics is presented in Figure 3.

As can be seen from our result, the counter statistics can assist request classification. For instance, while TPC-H query Q3, Q6, and Q12 all exhibit similar execution time, they vary in their need for memory bandwidth, making it easy to distinguish them if this statistic is used. Further, hardware statistics can also help project performance on new computing platforms. For example, by migrating to a machine with a faster processor but identical memory system, a memory-access-heavy request (TPC-H Q3) would show less performance improvement than one with lower bandwidth requirements (TPC-H Q6). Software metrics alone may not provide such differentiation.

3 Managing Resource Competition

Although many hardware metrics can be configured for observation, a processor usually has a limited number of counter registers to which the hardware metrics must map. Additionally, the configurations of some counter metrics are in conflict with each other and thus these metrics cannot be observed together. Existing mechanisms for processor counter statistics collection [1, 17, 20] do not support competing statistics collections simultaneously (*i.e.*, during the same period of program execution).

Competition for counter statistics can result from several scenarios. Since system functions typically require continuous collection of a specific set of hardware metrics over all program executions, any user-initiated counter statistics collection may conflict with them. In addition, on some resource-sharing hardware processors (particularly SMTs), sibling processors share the same

set of counter registers. Possible resource competition may arise when programs on sibling processors desire conflicting counter statistics simultaneously. Finally, a single application may request the use of conflicting counter statistics, potentially for different purposes.

In such contexts, processor counter statistics should be multiplexed for simultaneous competing uses and carefully managed for effective and fair utilization. The OS's basic mechanism to resolve resource competition is time-division multiplexing (alternating different counter register setups at interrupts) according to certain allocation shares. Time-division multiplexing of counter statistics has already been employed in SGI IRIX to resolve internal conflicts among multiple hardware metrics requested by a single user. Our focus here is to manage the competition between system-level functions and application-level tasks from multiple users.

The time-division multiplexing of hardware counter statistics is viable only when uses of counter statistics can still be effective (or sustain only slight loss of effectiveness) with partial-time sampled program execution statistics. Our allocation policy consists of two parts. First, the fraction of counter access time apportioned to system-level functions must be large enough to fulfill intended objectives but as small as possible to allow sufficient access to user-level tasks. Since the system-level functions are deterministic, we can statically provision a small but sufficiently effective allocation share for them. Second, when multiple user applications desire conflicting hardware counter statistics simultaneously (*e.g.*, when they run on sibling processors that share hardware counter registers), they divide the remaining counter statistics access time using any fair-share scheduler. We recognize that in order to be effective, certain applications might require complete monopoly of the hardware counters (*e.g.*, when a performance debugger must find out exactly where each cycle of the debugged program goes [1]). The superuser may change the default allocation policy in such cases.

Experimental results We experimentally validate that, as an important system-level function, the counter statistics-assisted CPU scheduler can still be effective with partial-time sampled counter statistics. Our CPU scheduler, proposed in Section 2.1, uses processor counter statistics to estimate each task's memory bus utilization. We install a once-per-millisecond interrupt which allows us to change counter register setup for each millisecond of program execution. In this experiment, the scheduling quantum is 100 milliseconds and we collect processor counter statistics only during selected millisecond periods so that the aggregate collection time accounts for the desired share for CPU scheduling.

Figures 4 and 5 show the scheduling results for SPEC-CPU2000 applications and server applications respectively, as the share of counter statistics collection time is varied. The experimental setup, metrics used, and applications are the same as in Section 2.1. In general, scheduling effectiveness depends on application behavior variability. Applications with high behavior variability may be difficult to predict even with more sophisticated non-linear table-based predictors [9]. However, for SPEC-CPU2000 applications, our results suggest that with only a 5% share of counter statistics collection time, the CPU scheduling performance and fairness approximates that attained with full counter statistics collection. For server applications, a 33% share is needed since the behavior of individual requests fluctuates and is more variable. These results indicate that the effectiveness of a counter statistics-assisted CPU scheduler may be fully realized with only a fraction of counter statistics collection time.

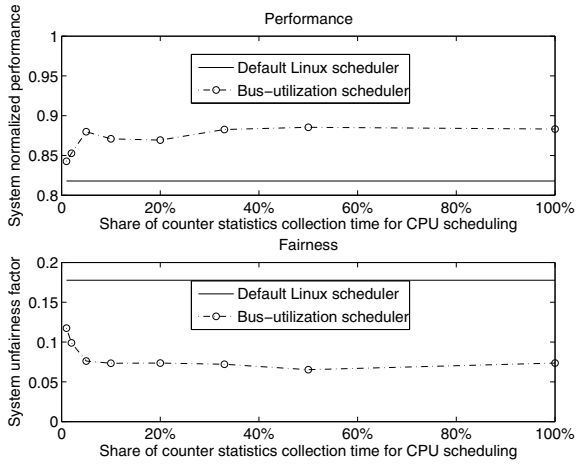


Figure 4: CPU scheduling performance and fairness of six SPEC-CPU2000 applications under partial-time sampled processor counter statistics. The "system normalized performance" and "unfairness factor" metrics were defined in Section 2.1.

4 Security Issues

Exposing processor counter statistics to non-privileged users and utilizing them in system functions may introduce new security risks. We examine two such issues.

Information leaks On resource-sharing multiprocessors, a program's execution characteristics are affected by behaviors of programs running on sibling processors. As a result, the knowledge of a program's own hardware execution characteristics may facilitate undesirable covert channel attacks [13] and side channel attacks for

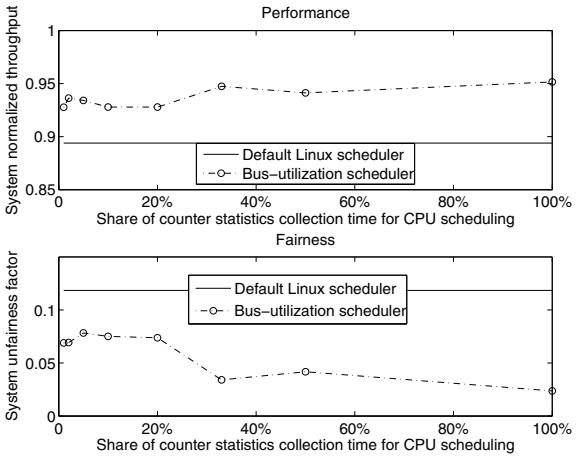


Figure 5: CPU scheduling performance and fairness of webclip and TPC-H applications under partial-time sampled processor counter statistics.

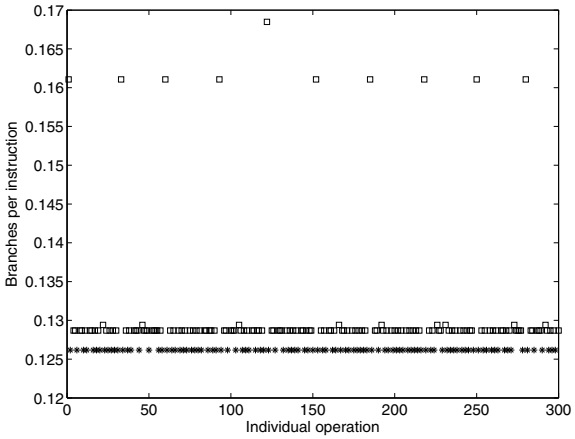


Figure 6: X^d is decomposed into a series of multiplications and squares in OpenSSL. Here X=1 and all d's are randomly generated with bit length no more than 6. Each point represents either a square (square symbol) or a multiplication (star symbol). The Y axis represents branches per instruction.

cryptanalysis [12].

Here we show a simple example of how the private RSA key in OpenSSL [14] may be stolen. One vital step in the RSA algorithm is to calculate "$X^d \bmod p$" where d is the private key and X is the input. In OpenSSL, modular exponentiation is decomposed into a series of modular squares and modular multiplications. By knowing the sequence of squares and multiplications, one can infer the private key d with high chance. For example, X^{11} is decomposed into $((X^2)^2 * X)^2 * X$. If one knows the execution order "sqr, sqr, mul, sqr, and mul", one can easily infer that the key is 11. Percival [16] showed that when a carefully constructed microbenchmark runs together with the RSA operation on the Intel Hyper-Threading platform, it may distinguish an RSA square from a RSA multiplication based on observed cache miss patterns. We show that this can be done more easily if certain hardware counter statistics are available. To demonstrate this, we run X^d exponentiation for many random private keys d and record the processor counter values for each individual multiplication and square operation. As we can see from Figure 6, the metric of branch instruction count divided by total instruction count (branches per instruction) provides a clear differentiation between the two types of operations.

The direct exposure of hardware counter statistics to non-privileged user applications may exacerbate existing risks in the following ways.

- Some statistics reported by hardware counters are combined event counts from multiple sibling processors (typically when the statistics concern shared hardware resources). Such statistics provide a direct way for a malicious program to learn information about other applications.
- Among those counter statistics that do not include event counts from sibling processors, some concern program execution characteristics that are affected by behaviors of programs running on sibling processors. An example is the L2 cache miss rate on L2 cache-sharing multiprocessors. These statistics can still be used to infer behavior patterns of other applications. Although such information may be learned at the software level (*e.g.*, through software sampling or probing), processor hardware counters expose the information with significantly higher accuracy while requiring almost no runtime overhead.

To prevent such undesirable information leaks, the system must be judicious in exposing hardware statistics to non-privileged user applications. Relevant hardware statistics can be withheld due to the above security concerns. Note that a fundamental tradeoff exists between security and versatility in information disclosure. In particular, withholding contention-dependent hardware statistics may impair an application's ability to employ adaptive self-management. Such tradeoffs should be carefully weighed to develop a balanced policy for hardware statistics exposure.

Manipulation of system adaptation When system functions utilize program execution characteristics for adaptive control, an application may manipulate such adaptation to its advantage by skillfully changing its execution behavior. For example, consider an adaptive CPU scheduler (like our bus-utilization scheduler proposed in Section 2.1) that uses partial-time sampled program execution characteristics to determine task resource usage levels and subsequently to run complementary tasks on

resource-sharing sibling processors. A program may increase its resource usage only during the execution statistics sampling periods in order to be unfairly classified as a high resource-usage task. A possible countermeasure for the adaptive scheduler is to collect program execution characteristics using randomized sample periods. In general, the OS should be aware of possible manipulations of system adaptation and design counter-measures against them.

5 Conclusion

With the increasing ubiquity of multiprocessor, multithreaded, and multicore machines, resource-aware policies at both the operating system and user level are becoming imperative for improved performance, fairness, and scalability. Hardware counter statistics provide a simple and efficient mechanism to learn about resource requirements and conflicts without application involvement. In this paper, we have made the case for direct OS management of hardware counter resource competition and security risks through demonstration of its utility both within the operating system and at user level. Ongoing work includes development of the API and policies for hardware counter resource management within the kernel.

References

[1] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Trans. on Computer Systems*, 15(4):357–390, November 1997.

[2] C.D. Antonopoulos, D.S. Nikolopoulos, and T.S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *Proc. of the 32nd Int'l Conf. on Parallel Processing*, October 2003.

[3] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35, HP Laboratories Palo Alto, 1999.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modeling. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[5] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proc. of the IEEE/ACM SC2000 Conf.*, Dallas, TX, November 2000.

[6] J.B. Bulpin and I.A. Pratt. Hyper-threading aware process scheduling heuristics. In *Proc. of the USENIX Annual Technical Conf.*, pages 103–106, Anaheim, CA, April 2005.

[7] I. Cohen, J.S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, pages 231–244, San Francisco, CA, December 2004.

[8] M. Curtis-Maury, J. Dzierwa, C.D. Antonopoulos, and D.S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proc. of the 20th Int'l Conf. on Supercomputing*, Cairns, Australia, June 2006.

[9] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, New Orleans, LA, September 2003.

[10] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proc. of the USENIX Annual Technical Conf.*, pages 395–398, Anaheim, CA, April 2005.

[11] A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer. Chip multithreading systems need a new operating system scheduler. In *Proc. of the SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[12] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2-3):141–158, 2000.

[13] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[14] OpenSSL: The open source toolkit for SSL/TLS. http://www.openssl.org.

[15] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical report, Department of Computer Science and Engineering, University of Washington, May 2000.

[16] C. Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, Canada, May 2005. http://www.daemonology.net/papers/htt.pdf.

[17] M. Pettersson. Linux performance counters driver. http://sourceforge.net/projects/perfctr/.

[18] A. Snavely and D. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *Proc. of the 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, Cambridge, MA, November 2000.

[19] P.F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behaviors of java applications. In *Proc. of the Third USENIX Virtual Machine Research and Technology Symp.*, pages 57–72, San Jose, CA, May 2004.

[20] Intel VTune™ performance analyzers. http://www.intel.com/software/products/vtune.

Microdrivers: A New Architecture for Device Drivers

Vinod Ganapathy, Arini Balakrishnan, Michael M. Swift and Somesh Jha
Computer Sciences Department, University of Wisconsin-Madison

Abstract

Commodity operating systems achieve good performance by running device drivers in-kernel. Unfortunately, this architecture offers poor fault isolation. This paper introduces microdrivers, which reduce the amount of driver code running in the kernel by splitting driver functionality between a small kernel-mode component and a larger user-mode component. This paper presents the microdriver architecture and techniques to refactor existing device drivers into microdrivers, achieving most of the benefits of user-mode drivers with the performance of kernel-mode drivers. Experiments on a network driver show that 75% of its code can be removed from the kernel without affecting common-case performance.

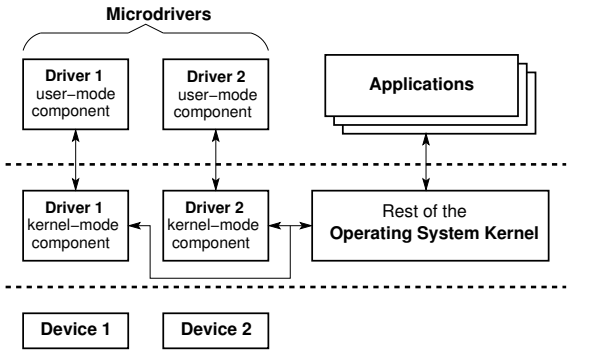


Figure 1: Microdrivers split device driver functionality between a kernel-mode component and a user-mode component.

1 Introduction

Bugs in device drivers are a major source of reliability problems in commodity operating systems. For instance, a recent Microsoft report indicates that as many as 85% of failures in Windows XP stem from buggy device drivers [19].

The root of the problem is the architecture of commodity operating systems. They are designed as *monolithic kernels* with all device drivers (and other kernel extensions), residing in the same address space as the kernel. This architecture results in good performance because invoking driver functionality is as easy and efficient as a function call. Unfortunately, this also results in bloated operating systems and poor fault isolation. For example, kernel extensions constitute over 70% of Linux kernel code [6] (a large fraction of these are device drivers), while over 35,000 drivers exist on Windows XP desktops [18]. A single bug exercised in any one of these extensions suffices to crash the entire operating system.

Several architectures have been proposed to isolate faults in device drivers [1, 9, 10, 16, 17, 22, 25]. For example, *microkernels* run device drivers as user-mode processes. A bug exercised in a device driver only results in the failure of the user-mode process running that driver. This approach, however, has two key problems. First, this architecture is not compatible with commodity operating systems, which are designed as monolithic kernels. Providing support for user-mode device drivers in commodity operating systems thus requires kernel modifications and rewriting device drivers [7, 14]. Second, switching between the kernel and a user-mode device driver involves the costly overhead of changing protection domains. For devices such as high-throughput network cards, this can result in significant latencies and performance penalties [22, 23]. A common escape hatch employed in such cases is to implement drivers within the kernel, which defeats the benefit afforded by microkernels.

This paper presents a new architecture for device drivers called *microdrivers*. Microdrivers seek the middle ground between monolithic kernels and microkernels, and improve reliability while maximizing performance. In a microdriver, the functionality of a device driver is *split* between a kernel-mode component and a user-mode component (Figure 1). The kernel-mode component contains critical and frequently used functionality, such as interrupt handling and performance-critical operations (*e.g.,* sending and receiving network packets and processing SCSI commands), while the user-mode component contains non-critical and infrequently used functionality (*e.g.,* startup/shutdown and error-handling). The user-mode component is implemented as a standalone process that is called from the kernel-mode component. Together, they provide the functionality of a traditional device driver.

Microdrivers are motivated by a simple mantra: *reduce the amount of code running in the kernel to improve*

its reliability. As discussed in Section 2, more than 70% of device driver code contains non-critical functionality. By relegating this code to a user-mode process, a microdriver reduces the amount of code running in the kernel and improves the reliability of the system as a whole. In addition, because the kernel-mode component of a microdriver is much smaller than the entire driver, it is amenable to verification and code audits.

Perhaps the most important aspect of microdrivers is compatibility with commodity operating system architectures—device drivers written for monolithic kernels can be *refactored nearly automatically* into microdrivers. This provides a path to execute device drivers written for commodity operating systems in user-mode without sacrificing performance. Prior efforts at user-mode device drivers have either required rewriting the driver completely [7, 14] or impose significant performance penalty [23]. We show that program analysis techniques can automatically identify critical functions in a device driver, following which a semantics-preserving transformation can split the driver into a user-mode and a kernel-mode component. We discuss the design and implementation of such a refactoring tool in Section 4. We used this tool to refactor the e1000 device driver for the Intel PRO/1000 gigabit network adapter into a microdriver. The kernel-mode component of this microdriver contains just 25% of the code of the entire microdriver. Our preliminary experience with this microdriver indicates that overheads for common-case performance and CPU utilization are negligible.

2 The case for microdrivers

Because device drivers communicate with I/O devices, their performance is critical to ensure fast I/O. Conventional wisdom holds that performance-critical functionality must be implemented in the kernel. Even undergraduate texts preach that I/O algorithms must be implemented in the kernel for good performance [21, page 427]. Unfortunately, a popular interpretation of this tenet is that device drivers must reside in the kernel. This has lead to the monolithic and unreliable operating systems that we see today.

Surely, the *entire driver* does not reside on the performance-critical path? To answer this question, we conducted a study of 455 device drivers, comprising network, SCSI and sound drivers from the Linux 2.6.18 kernel, and identified performance-critical functions in each of them. To do so, we extracted the static call-graph of each driver—this graph has an edge $f \rightarrow g$ if function f can potentially call function g. We resolved calls via function pointers using a simple pointer analysis that is conservative in the absence of type-casts—each function pointer can resolve to any function whose address is taken, and whose type signature matches that of the function pointer.

Driver family	Drivers analyzed	Critical funcs
Network	134	27.8%
SCSI	49	26.1%
Sound	272	7.8%

Figure 2: Classification of functions in different families of Linux device drivers.

We then identified a set of *critical root functions* that are driver entrypoints that must execute in the kernel for high performance. Critical root functions are those that handle interrupts or execute at other high-priority levels (*e.g.,* tasklets, bottom-halves), and functions that supply data to or receive data from a device. We define performance-critical functions to be critical root functions plus the functions that they transitively call. Given a template of the entrypoints, critical root functions can be identified automatically for each family of drivers: *e.g.,* functions that transmit network packets are critical for network drivers, while functions that process SCSI commands are critical for SCSI device drivers. We wrote a tool to automatically identify critical root functions (based upon their type signatures) and the functions that they transitively call.

Figure 2 shows the results of our study. We found that fewer than 30% of the functions in a driver are performance critical. The remaining functions are called only occasionally, *e.g.,* during device startup/shutdown, to configure device parameters, and to obtain diagnostic information. Consider, for example, the e1000 driver for the Intel PRO/1000 gigabit network adapter, one of the drivers considered in our study. Critical root functions for this driver include the interrupt handler, the function to transmit network packets, and callback functions registered with the kernel to poll the device. This driver contains 274 functions containing approximately 15,100 lines of source code. Of these, just 25 functions containing approximately 1,550 lines of source code were classified as critical. It suffices to execute just these functions in the kernel for good performance. Relegating the remaining functions, which handle startup/shutdown and get/set device parameters, to a user-mode process will greatly reduce the amount of code running in kernel space without adversely impacting common-case performance. Note that our estimate of critical device driver code is conservative, because we only identify critical functions. It is likely that a finer-grained approach will show that even less code is on the critical path.

Three factors lead us to believe that implementing non-critical functionality as a user-mode process will also improve system reliability and availability as a whole.

First, fault isolation improves. Any bugs that crash the user-mode process of a microdriver will potentially render the corresponding device unusable but will not affect the rest of the operating system. The device driver can then be restarted in isolation to restore operation of the device. Note that because the kernel-mode component of a microdriver implements critical device functionality, such as interrupt processing, it is possible to keep the device operational even if the user-mode process crashes. For example, the kernel-mode component can implement error-checking code that detects that the user-mode process has crashed, and wait until the process restarts. However, as it does so, it can still serve other requests to/from the device.

Second, because the kernel-mode component of the microdriver implements critical and heavily-used functionality, this code is likely more heavily tested than the user-mode component. Further, because the kernel-mode component is a relatively small entity, it can either be verified, subject to thorough code audits, or be protected with mechanisms such as SFI [24].

Third, because the kernel-mode component and the user-mode component of a microdriver communicate via a narrow interface (as desribed in Section 3), data passed between the kernel- and user-mode components can be sanity-checked for errors. For example, a bug in the user-mode component may introduce a dangling pointer in a data structure that it then passes to the kernel. However, the corrupted data structure can be detected using error-checking code implemented at the interface, thus potentially preventing corruption of kernel data structures.

Indeed, the tenet that reduced code in the kernel means improved reliability has also been recognized by many others [4, 7, 9, 13], and is an important motivation for microkernels. This has resulted in several services, that were previously implemented in the kernel, being implemented in user-mode (*e.g.,* kernel module loaders, AFS). There have also been several recent efforts to redesign device drivers (in particular, graphics drivers) with the goal of reducing the amount of code running in the kernel [4, 13].

Finally, microdrivers also allow vendors to take advantage of user-level tools such as profilers and debuggers during the driver development process. Comparable tools for developing kernel code are fewer in number and not as advanced because kernel programming represents a smaller market and a more challenging target.

Of course, microdrivers are not a panacea for device driver reliability problems. A bug in the kernel-mode component of a microdriver could still crash the operating system. It is also possible that by splitting functionality between a user-mode and kernel-mode component, microdrivers can expose otherwise latent bugs. For example, a latent race condition in a device driver could potentially be exposed in its microdriver implementation.

3 Architecture of a microdriver

A microdriver consists of a kernel-mode component that implements critical functionality and a user-mode process that implements non-critical functionality. Device driver functionality is split between the kernel-mode and user-mode components at function boundaries. The two components communicate using an LRPC-like mechanism [3]. In the kernel-mode component, direct calls to functions implemented in the user-mode component are replaced with upcalls through stubs. Stubs marshal data structures accessed by the called function and unmarshal them when the call returns. A symmetric downcall mechanism enables the user-mode component to invoke kernel functions. To handle multiple requests from the kernel-mode component, the user-mode process is multithreaded.

An object tracker, similar to the one used by Nooks [22], synchronizes copies of a data structure in the kernel's address space and the user-mode process' address space. It has three main functions.

First, the object tracker is invoked during marshaling/unmarshaling to translate pointers between address spaces. This ensures that updates to a data structure in one address space are reflected on its copy in the other address space. Doing so is challenging for complex data structures such as arrays, whose elements are accessed as offsets from the start of the data structure. The object tracker must explicitly store the range of such data structures and ensure that accesses via offsets are translated correctly between address spaces.

Second, the object tracker ensures that data structures allocated/deallocated in one address space are also allocated/deallocated in the other. Allocations are dealt with during pointer translation—a new data structure is allocated and initialized in an address space if no corresponding copy is found in that address space. Dealing with deallocations is more challenging. Deallocator functions must update the object tracker's database by removing the entry for the data structure being deallocated.

Third, the object tracker manages synchronization of shared data structures. Two copies of a shared data structure can exist in a microdriver, one in each address space, only one of which must be modified at any time. To support concurrent accesses to such data structures, the user-mode process must synchronize with the kernel to acquire a lock on a shared data structure. The object tracker must ensure that any updates to a shared data structures in one address space are reflected to its copy in the other address space.

In addition to the basic functions described above, the object tracker can optionally include error-checking code

to check for a variety of common data structure corruptions, such as dangling pointers and potential null-pointer dereferences.

Several enhancements are possible to the basic architecture of a microdriver. Functions that are repeatedly called from both the kernel-mode component and user-mode component can potentially be replicated in both components, thus avoiding the overhead of an address-space change each time the function is accessed. Similarly, a frequently-accessed data structure can be allocated in a shared memory region that is accessible both to the kernel and the user-mode process. Finally, to ensure fast operation of the user-mode process, the operating system can pin the process' pages to memory.

4 Refactoring device drivers to microdrivers

Microdrivers present the same interface to the kernel as traditional device drivers, and are thus compatible with commodity operating systems. Moreover, code to implement upcalls, downcalls, marshaling and unmarshaling follows a standard template and can be automatically generated. This section presents the design of a tool that statically refactors traditional device drivers into microdrivers (see Figure 3). Such a tool preserves the investment in existing device drivers and provides a migration path to create microdrivers.

The refactoring tool has two functions. First, it must analyze the device driver and determine which functions are critical. This is achieved by the *splitter*. Second, it must move the remaining (non-critical) functions to a user-mode component, and generate code for communication between the kernel-mode and user-mode components. This is achieved by the *code generator*.

The splitter. The splitter analyzes the device driver and determines how functions implemented in the driver must be split between kernel-mode and user-mode. It builds a static call-graph of the driver (including edges for indirect calls), identifies critical root functions, and classifies functions transitively called by them as critical, as described in the study in Section 2. Critical root functions need to be identified just once for each family of device drivers.

While this simple propagation-based approach to identify critical functions has worked well for drivers that we have considered so far, the splitter can employ more sophisticated algorithms that use dynamically gathered profile information. For example, critical functions can be inferred by solving an optimization problem on the static call-graph modeled as a *flow network* [2] with weights on edges and nodes. Edge weights denote call frequencies (obtained by profiling) and node weights are proportional to the number of lines of code in the function denoted by the node. The goal is to find a cut in the graph under the constraint that all nodes representing critical root functions must appear on one side of the cut (the critical side). Additional constraints can also be imposed, *e.g.,* that a critical section must not be split between the kernel-mode and user-mode components. Further, the cut should be optimal: it should mimimize both the weight of edges crossing the cut and the weight of nodes on the critical side of the cut. All nodes on the critical side of the cut are marked critical, and the remaining nodes are non-critical. Intuitively, such a cut minimizes the number of switches between protection domains and also the amount of code running in the kernel.

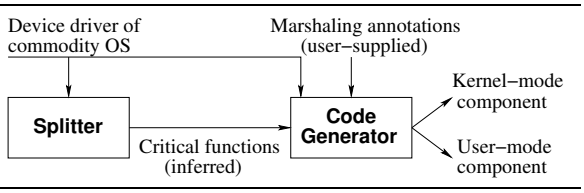


Figure 3: Design of a tool to refactor traditional device drivers into microdrivers.

The code generator. The code generator uses the critical functions identified and emits code for the kernel-mode and user-mode components. It also generates all the code to handle upcalls and downcalls, including stubs and code to marshal/unmarshal data structures. The object tracker and threadpool implementation (for the multithreaded user-mode component) are common to all microdrivers and need to be written just once.

The code generator needs *marshaling annotations* to guide the generation of marshaling/unmarshaling code. These annotations are used to specify the length of dynamically allocated arrays, linked lists and other complex data structures. The code generator employs a conservative static analysis algorithm to identify variables that represent such data structures and prompts the user to provide these annotations. This potentially reduces the traditional burden associated with annotation, because the user does not have to provide annotations beforehand, but only as guided by the code generator, and only for data structures that cross the user/kernel boundary. For example, for the e1000 device driver, the code generator automatically infers that variables of type `struct e1000_rx_ring` and `struct e1000_tx_ring` (among others) are arrays. These denote ring buffers that are allocated by the e1000 driver at startup. It requests marshaling annotations for each function call that crosses address-spaces and potentially modifies these data structures. The user must supply marshaling annotations that determine the length of these data structures (or supply predicates, *e.g.,* those that determine how to stop traversing a linked list).

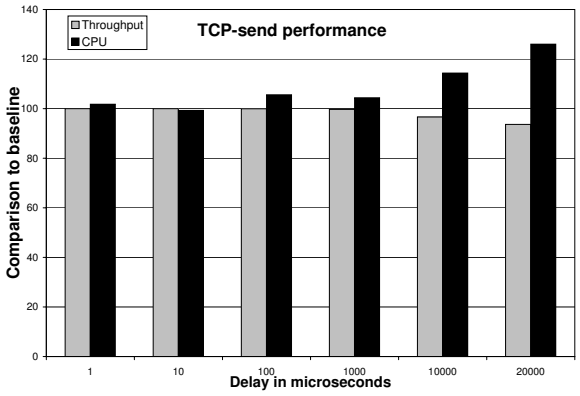


Figure 4: Performance of an e1000 microdriver.

5 Implementation and experiments

We have implemented several portions of the microdriver architecture and the refactoring tool. In particular, the refactoring tool automatically identifies a split and generates code for the kernel- and user-mode components. We have also implemented a static analysis algorithm to infer where marshaling annotations are necessary and are currently in the process of integrating this with the code-generator for marshaling/unmarshaling. To date, we have applied the tool to several network drivers. Because our infrastructure is still in development, we report our experience simulating the operation of an e1000 microdriver. In particular, we used our tool to generate code for the kernel- and user-mode components, and ran both the components in the kernel address-space, using delays to simulate change of protection domains.

The kernel-mode component of our e1000 microdriver contains just 25% of the code of the entire microdriver. In our experiments, we ran the e1000 microdriver on a dual-core 3Ghz Pentium-D machine running Linux-2.6.18. We measured network throughput and CPU utilization using `netperf` to send TCP packets (results for TCP/receive were similar and are omitted). We used buffers of size 256KB on both the sending and receiving side and transmitted 32KB messages. Figure 4 compares the network throughput and CPU utilization of the e1000 microdriver (with different values for delays) against a traditional e1000 device driver running under the same conditions. We observed that the microdriver has negligible overheads for network throughput and CPU utilization for delays under 10μs. Even with a 20ms delay ($60,000,000$ machine cycles) we only observed a 6.3% drop in network throughput and 26% increase in CPU utilization. These results show that microdrivers reduce the amount of driver code running in the kernel without affecting common-case performance, and are thus a viable alternative to traditional device drivers.

6 Related work

Hardware-based isolation. Several architectures use hardware-based mechanisms to isolate faults in kernel extensions, in particular device drivers. These include Nooks [22] and VMM-based mechanisms [11, 15] that run each driver in its own protection domain. Microdrivers also use hardware, in particular, the process boundary, to isolate large parts (but not the entire) device driver. However, microdrivers can potentially perform better than these hardware-based isolation mechanisms because performance-critical code resides and executes in kernel address-space. In addition, microdrivers also reduce the amount of code running in the kernel. Microkernels (*e.g.,* [16, 23, 25]) also use the process boundary to isolate device drivers, and explicitly aim to reduce the amount of code executing with kernel privilege, but do so at the cost of reduced performance. Microdrivers offer poorer isolation than microkernels, but promise better performance.

Several recent efforts have focused on reducing the amount of driver code running in commodity operating system kernels [4, 7, 8, 13, 14, 17]. The FUSD framework [8] and the Microsoft user-mode driver framework [17] offer APIs to program user-mode device drivers that communicate with a kernel module. Chubb [7] and Leslie *et al.* [14] report user-mode driver performance comparable to in-kernel device drivers. However, unlike microdrivers, they require both kernel support, and rewriting device drivers, making them incompatible with existing operating systems.

Language-based isolation. SafeDrive [27] is a recent effort to improve device driver reliability by preventing type safety violations (and is similar in spirit to SFI [24]). SafeDrive reports good performance and is compatible with commodity operating systems. However, unlike microdrivers, SafeDrive does not reduce the amount of in-kernel code. Moreover, it does not offer protection against bugs that do not violate type safety (*e.g.,* violation of the locking protocol or other kernel API usage rules).

Program partitioning. Automatic and semi-automatic program partitioning techniques, much like the ones in Section 4, have also been applied to improve application security [5, 26] and to improve the performance of distributed components [12] and data-intensive user applications [20].

7 Conclusions

Microdrivers are a promising alternative to existing architectures for device drivers. They can improve system reliability by reducing the amount of code running in the kernel without adversely affecting common-case performance. Because microdrivers are compatible with commodity operating systems, they offer a path for running

existing device drivers in user-mode with good common-case performance. This paper also shows that program analysis and transformation techniques can refactor existing drivers nearly automatically into microdrivers.

References

[1] The Minix3 operating system. `www.minix3.org`.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, February 1993.

[3] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM TOCS*, 8(1), February 1990.

[4] D. Blythe. Windows graphics overview. In *Windows Hardware Engineering Conference*, 2005.

[5] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *13th USENIX Security Symposium*, 2004.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *18th ACM SOSP*, 2001.

[7] P. Chubb. Get more device drivers out of the kernel! In *Ottawa Linux Symposium*, 2004.

[8] J. Elson. FUSD: A Linux framework for user-space devices, 2004. User manual for FUSD 1.0.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *15th ACM SOSP*, 1995.

[10] A. Forin, D. Golub, and B. Bershad. An I/O system for Mach. In *USENIX Mach Symposium*, 1991.

[11] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[12] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *4th ACM/USENIX OSDI*, 1999.

[13] B. Langley. Windows "Longhorn" display driver model—details and requirements. In *Windows Hardware Engineering Conference*, 2004.

[14] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5), September 2005.

[15] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th ACM/USENIX OSDI*, 2004.

[16] J. Liedtke. On μ-kernel construction. In *15th ACM SOSP*, 1995.

[17] Microsoft. Architecture of the user-mode driver framework, May 2006. Version 0.7.

[18] B. Murphy and M. R. Garzia. Software reliability engineering for mass market products. *The DoD Software Tech News*, 8(1), 2004.

[19] V. Orgovan and M. Tricker. An introduction to driver quality. Microsoft WinHec 2004 Presentation DDT301, 2003.

[20] A. Purohit, C. P. Wright, J. Spadavecchia, and E. Zadok. Cosy: Develop in user-land, run in kernel-mode. In *9th HotOS*, 2003.

[21] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. Addison Wesley, fifth edition, 1999.

[22] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM TOCS*, 23(1), February 2005.

[23] K. T. Van Maren. The Fluke device driver framework. Master's thesis, Dept. of Computer Science, Univ. of Utah, December 1999.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th ACM SOSP*, 1993.

[25] M. Young, M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and A. Tevanian. Mach: A new kernel foundation for UNIX development. In *Summer USENIX Conference*, 1986.

[26] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM TOCS*, 20(3), August 2002.

[27] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th OSDI*, 2006.

MashupOS: Operating System Abstractions for Client Mashups

Jon Howell howell@microsoft.com
Collin Jackson collinj@cs.stanford.edu
Helen J. Wang helenw@microsoft.com
Xiaofeng Fan xiaoffan@microsoft.com

Abstract— **Web browser support has evolved piecemeal to balance the security and interoperability requirements of client-side script services. This evolution has led to an inadequate security model that forces Web applications to choose between security and interoperation. We draw an analogy between Web sites' sharing of browser resources and users' sharing of operating system resources, and use this analogy as a guide to develop protection and communication abstractions in MashupOS: a set of abstractions that isolate mutually-untrusting web services within the browser, while allowing safe forms of communication.**

I. Introduction

Web browsers are becoming the single stop for everyone's computing needs including information access, personal communications, office tasks, and e-commerce. Today's Web applications synthesize the world of data and code, offering rich services through Web browsers and rivaling those of desktop PCs. Browsers have evolved to be a multi-principal operating environment where mutually distrusting Web sites (as principals) interact programmatically in a single page on the client side, sharing the underlying browser resources. Consider a scenario (Figure 1) wherein an HTML file (possibly including scripts) sent from `webmail.com` and an HTML file sent from `stocks.com` run on the client browser. These HTML files are really delegates on behalf of `webmail.com` and `stocks.com`, respectively, using resources on the client to improve the interactivity of the services. In this scenario, the sites are mutually distrusting principals sharing the browser's resources (display, memory, CPU, network access). This resembles the PC operating environment where mutually distrusting users share host resources.

However, unlike PCs that utilize multi-user operating systems for resource sharing, protection, and management, today's browsers do not employ any operating system abstractions, but provide just a limited binary trust model and protection abstractions suitable only for a single principal system: There is either *no trust* across principals through complete isolation or *full trust* through incorporating third party code as libraries. The browser abstraction for the former is FRAME; frames enable interactive (script-enhanced) Web services to occupy neighboring display real estate, but the components

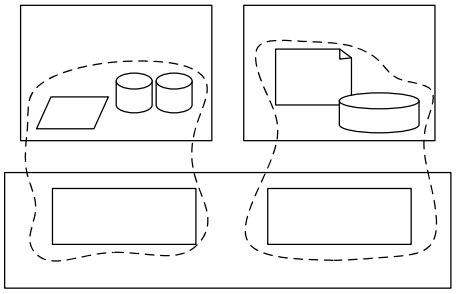


Fig. 1. JavaScript code running on a client browser is really just a distributed component of the Web service that provided the code.

cannot interact. The abstraction for the latter is SCRIPT which allows third-party scripts to be included as library code; the embedded cross-domain scripts enjoy full trust from its includer and can access the includer's data, display, and access to back-end server resources. With these limited existing browser abstractions, Web programmers are forced to make tradeoffs between security and functionality, and often times sacrifice security for functionality. In the scenario above, a web program either segregates HTML content from `webmail.com` and `stocks.com` into separate frames denying any communications or embed their scripts as libraries into a containing page allowing intimate interactions. As we can see, controlled interactions may be desired: If the `stocks.com` server offers a limited Web interface that other servers such as `webmail.com` may access, then the browser should allow similar communication between the corresponding components running on the client. This controlled communication among otherwise isolated client components is not attainable in today's browsers.

In the MashupOS project, we aim to design and build a browser-based multi-principal operating system. In this position paper, we outline our initial explorations on the proper abstractions needed for protection and communciations which to date have received only ad-hoc band-aids and patches. By identifying an appropriate, strong analogy to conventional operating system design,

MashupOS draws on decades of wisdom and experience in managing isolation and communication among untrusted principals. While there is no way to show an open architecture to be complete and correct, the MashupOS approach has a solid foundation.

For the rest of the paper, Section II identifies the limitations of the abstractions and security policies implemented in contemporary browsers. Section III sets out the goals of MashupOS. Section IV introduce primitive abstractions which plainly enforce a security model analogous to a multi-principal operating system, providing only spare communication primitives. Section V enhances those primitives with syntactic sugar, providing familiar, simple communication without destroying the security of the primitives. Section VI considers deployment issues, and Section VII relates MashupOS to other proposals. Finally, we conclude in Section VIII.

II. Background

The security policy of current browsers is the result of a patchwork of decisions made by many independent companies and individuals, with a heavy emphasis on avoiding vulnerabilities in legacy sites, rather than providing the best abstractions for the newest sites. To motivate our proposal, we examine the security policies of current browsers and the limitations that they place on Web mashups.

A. Same-Origin Policy

Browsers use cookies as a way to identify and authenticate unique users, and to operate in a way that depends on which user is viewing the page. A cookie is a small, arbitrary piece of data chosen by the Web server and sent to the browser in an HTTP header when responding to a request. On subsequent requests, browsers use HTTP headers to echo back cookies to the server that sent them [8].

In order for cookies to be used as an authentication mechanism and provide the illusion of an isolated session shown in Figure 1, the browser must keep cookies secret from other sites. Thus, cookies are sent only to the same site that set them, a policy known as the *Same-Origin Policy* [6].

B. AJAX

Recently, the *AJAX* programming model has emerged, allowing web services to shift interactive user interface code from the web server to the browser. AJAX stands for Asynchronous JavaScript and XML. Where conventional web pages handle every click with a round-trip to the server, AJAX uses client-side code ("JavaScript") to handle many user interactions, providing interactivity not bounded by network and server performance. Furthermore, when communication with the server is required, that communication occurs asynchronously ("asynchronous XML"), while the client-side code continues to provide interactivity in the meantime.

The Document Object Model (DOM) is an interface that allows scripts to read and modify HTML documents, even documents in other pages or frames. To ensure that web pages cannot circumvent firewalls or hijack the user's authenticated sessions, JavaScript documents loaded from one origin are prevented from getting or setting properties of a document from a different origin. Each browser window, FRAME, or IFRAME (inline frame) is a separate document, and each document is associated with an origin on the basis of URL. Two origins are considered separate if they differ by scheme (http or https), DNS name, or TCP port [12]. For example, frames from `http://amazon.com/` and `http://amazon.co.uk/` cannot access each other's resources because their DNS names differ.

The asynchronous XML communication of AJAX is accomplished using XMLHttpRequest, which can communicate only with the page's origin server. For example, a frame from `http://amazon.com/` cannot issue an XMLHttpRequest to `http://amazon.co.uk/`.

C. Remote Code Inclusion

Web developers often wish to incorporate third-party code libraries. An example is housingmaps.com, which uses the Google Maps code library to visualize Craigslist housing classified ads. Because JavaScript files generally do not have any user-specific or sensitive information, browsers interpret files in this format as public code libraries and allow them to be executed across domains. The code runs with the privileges of the page including it. For example, the `housingmaps.com/index.html` page may contain the markup `<script src='http://google.com/maps.js'> </script>`, which allows `maps.js` to access housingmaps.com's HTML DOM objects, cookies and data through XMLHttpRequest. However, `maps.js` cannot access google.com's resources since the code in `maps.js` is considered to have the origin housingmaps.com rather than google.com in this context.

The existence of remote code inclusion as an alternative to the isolation of the Same-Origin Policy presents web developers with a dilemma: a site must either completely distrust another site and segregate itself through the use of cross-domain frames, or a site can use another site's code as its own, offering full resource access to the remote site.

D. Web Mashups

Web mashups compose data from more than one site, yet the browser prevents such cross-domain communica-

tion. XMLHttpRequest cannot retrieve data from another domain, even if the other domain desires it.

Initially, mashup developers worked around these restrictions using a proxy approach: a web portal like MSN.com may, in the back end, compose dozens of web services together into a single web page which is displayed in the browser. Services from different principals can be composed the same way; examples include metacrawlers and news aggregators. This approach unfortunately makes several unnecessary round trips, reducing performance, and the proxy can become a choke point, limiting scalability.

An alternative to proxies is encoding public data in executable JavaScript format (JavaScript Object Notation, or JSON [2]). Using SCRIPT tags, this data can be passed from the provider to the integrator across domain boundaries, eliminating the need for proxies. As a possibly unintended side effect, this technique grants the integrator's privileges to the data provider.

E. Gadget Aggregators

Mutually distrusting network servers communicate with one another using web service APIs, but the Same-Origin Policy offers no equivalent communication among client-side components. As a result, web services that wish to enjoy tight client-side coupling must abandon entirely the isolation afforded by the policy, and instead compose scripts directly using the SCRIPT tag. Scripts composed this way can communicate because all the scripts are treated as belonging to the domain of the enclosing document. Unfortunately, that communication is completely unconstrained. Two examples of compositions follow in which such trust is inappropriate.

Web gadget aggregators such as Google Personalized Homepage page [5] and Windows Live [9] aggregate user-selected interactive content from disparate sources into a single portal page. A *gadget* includes both HTML and JavaScript, and is designed to be included into a gadget aggregator page; it is the client-side of some web service.

Gadget aggregators are security-conscious; third-party gadgets are hosted on a separate domain and IFRAMEs are employed to isolate these gadgets from one another and from the containing page. However, because these IFRAMEs cannot communicate, aggregators also support *inline* gadgets, SCRIPTs inlined directly into the aggregator page. Because inlining requires complete trust, Google's aggregator asks the user what to do: "Inline modules can ... give its author access to information including your Google cookies and preference settings for other modules. Click OK if you trust this module's author."

AOL provides a chat widget, called Web AIM, designed to be integrated into other web services' displays. A web service can interoperate with the widget using an API to add contacts. Ideally, the service should not be allowed arbitrary access to the widget, with which a malicious service might extract the user's list of contacts or even feign a chat session. Absent better abstractions, AOL chooses interoperability, and punts security to the user with a "click OK" dialog box.

III. Goals

Secure, interoperating client-side service compositions demand new browser abstractions with three properties.

- **Cross-domain protection** prevents code in one domain from compromising the confidentiality or integrity of other domains. To prevent denial of service, domains should receive fair shares of commodity resources such as CPU, network bandwidth, and disk space.
- **Controlled cross-domain communication** allows a service from one domain to interoperate with a service from another, enabling rich composition.
- **Doing minimal violence** to the existing Web API eases adoption of the new abstractions, and the mechanisms must offer acceptable backwards-compatibility behavior.

IV. Primitive Abstractions

Section IV-A identifies the resources MashupOS manages. The ServiceInstance of Section IV-B isolates principals from each others' resources, and the abstractions of Section IV-C provide a restricted communication model among ServiceInstances.

A. Resources

Reusable commodity resources, such as CPU, memory, and network bandwidth, need only be fairly shared to prevent misbehaving principals from denying service to well-behaved principals.

Unique resources, however, require access control abstractions to misbehaving principals from violating confidentiality or integrity. **Persistent storage** in the browser is much simpler than conventional OS file systems; therefore, rather than extend the OS analogy to share persistent state among principles, MashupOS maintains the isolated persistent storage model of today's browsers. Likewise, MashupOS offers no shared **memory** between principals, so no memory access-control abstraction is necessary.

In MashupOS, the **display** is an access-controlled resource in the same sense that the X11 window server enables mutually-distrusting clients to share access to a common display resource.

Likewise, **network access** from client code is subject to access controls that mimic those enforced on accesses originating at the principal's web server, following the

intuition of Figure 1. Indeed, communication between client components is treated as network communication, and subject to the same access controls.

B. The ServiceInstance Isolation Primitive

We propose a primitive abstraction called a SERVICEINSTANCE, analogous to an operating system process. A new browser window is initially associated with a single SERVICEINSTANCE, which contains the page's Document Object Model (DOM) structure and the ephemeral state (variables) associated with the code on the page. The SERVICEINSTANCE is also associated with a single principal, defined as the Same-Origin Policy domain associated with the document loaded into the page.

Just as in today's HTML, if a SERVICEINSTANCE uses a SCRIPT tag to incorporate a trusted library by reference, that library runs as the principal associated with the SERVICEINSTANCE, regardless of the location from which the script was fetched. The same effect can be achieved if the SERVICEINSTANCE principal's server fetches the script and inlines it into the SERVICEINSTANCE's HTML: in both cases, the SERVICEINSTANCE is owned entirely by the principal that served the outer HTML, and the inlined script is completely trusted to run on behalf of that principal.

We introduce a new HTML element `<FRIV src="page2.html">` that crosses the security isolation of a FRAME with the layout and communications benefits of a DIV. The FRIV has three effects. First, it allocates a subregion of the outer display region. Second, it creates a new SERVICEINSTANCE. Third, it populates the DOM of the subregion by loading the referenced SRC document.

The DOM that represents the content of the subregion is isolated inside the new SERVICEINSTANCE. Script code in the inner SERVICEINSTANCE cannot reach the DOM of the outer SERVICEINSTANCE by traversing pointers (parent pointers, callbacks, or anything else), nor vice versa. Likewise, the ephemeral state of the scripts loaded into the inner SERVICEINSTANCE are isolated: the inner SERVICEINSTANCE cannot hold a reference to objects in the outer SERVICEINSTANCE, nor vice versa.

The outer SERVICEINSTANCE (that created the FRIV) and the inner SERVICEINSTANCE divide responsibility over the display resource. The outer SERVICEINSTANCE is responsible for all of its display other than the contents of the FRIV display region, and the inner SERVICEINSTANCE is responsible for the contents of the FRIV. The outer SERVICEINSTANCE's DOM represents the delegated display region with a "top-half FRIV" object: the outer SERVICEINSTANCE can adjust the display area of the FRIV, but cannot see the composite DOM below that point. Likewise, the inner SERVICEINSTANCE's root DOM object is a "bottom-half FRIV" object: the inner SERVICEINSTANCE can read the size of the allocated region, and populate it with DOM elements, but cannot see the composite DOM elements above the FRIV boundary.

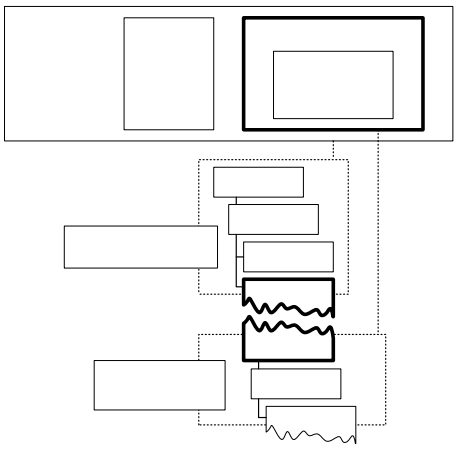


Fig. 2. The diagram on top shows the subdivision of a display surface into regions. Dotted boxes on the below represent the SERVICEINSTANCEs that contain the DOM elements and other private data associated with each display region. The two SERVICEINSTANCEs shown adjoin by a FRIV object, highlighted by a bold line. Each SERVICEINSTANCE can only access its "half" of the FRIV object. The FRIV represents the subdivision of display space (shown by the bold boundary in the top diagram) and provides an explicit, data-only communication channel between the SERVICEINSTANCEs.

FRIV can be used recursively: a FRIV may contain another FRIV. For example, a gadget aggregator (in a FRAME) may contain a mail-reading gadget (in a FRIV), which may recursively contain a content-viewing pane (in another FRIV).

C. Communication among SERVICEINSTANCEs

The FRIV is the DOM object that connects two SERVICEINSTANCEs: The outer SERVICEINSTANCE delegates control over part of its display to the inner SERVICEINSTANCE. Each SERVICEINSTANCE is a represention on the client of some web site; sites may reasonably wish to communicate in some limited (mutually-distrusting) fashion between SERVICEINSTANCEs. Therefore, the FRIV includes an explicit communication channel between the SERVICEINSTANCEs it joins.

The inner SERVICEINSTANCE can write messages into the bottom-half FRIV, and the outer SERVICEINSTANCE can register a callback with the top-half FRIV to receive such messages. The messages are data-only, one-way messages, and therefore have semantics equivalent to discrete network messages between the servers that are responsible for the SERVICEINSTANCE content. In particular, data-only messages mean that a SERVICEINSTANCE cannot hang itself by giving away a pointer to its DOM or internal state. Our choice of

an asynchronous communication abstraction ensures that SERVICEINSTANCEs can communicate without failure-coupling to one another; AJAX-style asynchronous RPC is easy to build over asynchronous messages.

The primitive syntax for explicit communication channels emphasize the simplicity of the channel, in particular its equivalence to a network channel: Each half-FRIV object contains an `registerReceiveHandler(handler)` method, which registers a JavaScript function to receive data, and a `send(data)` method, which sends data to the handler on the other end of the channel. The receiver also learns the identity of the principal that sent the message.

The MashupOS implementation allows messages to contain only value types (including compounds). Because messages may *not* transmit a pointer to data in the sending SERVICEINSTANCE, they behave just as a network message between the sites on behalf of which the SERVICEINSTANCEs run.

D. Commodity Resource Isolation

Given the SERVICEINSTANCE isolation abstraction, we can apply conventional techniques to fairly share commodity resources such as CPU, memory, and network bandwidth. This keeps a resource-hog from breaking other services.

For pedagogy, we described each SERVICEINSTANCE as associated with exactly one display region. In practice, we expect to generalize the SERVICEINSTANCE to manage zero or multiple display regions, providing the analog of background processes and multi-window applications.

E. *Null-Principal* SERVICEINSTANCE*s*

We include as an optional enhancement the ability for a principal to create a SERVICEINSTANCE with a null principal. Such a SERVICEINSTANCE receives the usual fair share of commodity resources, and may also draw into its FRIV and communicate with its creator via its FRIV. Because the SERVICEINSTANCE has no principal, it cannot make XMLHttpRequests; any JSONRequests carry the null principal as their origin. Cookies, which are associated with principals, are unavailable to the null-principal SERVICEINSTANCE. The email gadget display can use a null-principal SERVICEINSTANCE to render untrusted email content fetched from `webmail.com` without giving that content access to the principal's resources.

V. SYNTACTIC SUGAR ABSTRACTIONS

The FRIV primitive described in Section IV is like today's cross-domain FRAME, plus commodity resource isolation and a message-based communication channel. Developers, however, prefer `DIV`s to `FRAME`s for two reasons. First, `DIV`s better negotiate display real estate to accomodate documents of varying size. Second, communication between the gadget charged with implementing a DIV and the containing document is as simple as method invocation. In this section, we enhance the FRIV to exhibit these preferred behaviors. These enhancements do not weaken the primitive model; indeed, it is possible to implement them as syntactic sugar over the primitive model.

One advantage of using a DIV rather than a FRAME to contain unknown content is that DIVs automatically resize to fit their content, whereas FRAMEs force the user to drag through the content with a scrollbar. A FRIV is rendered like a frame by the browser, to prevent the inner gadget from using absolute positioning to display elements outside the FRIV's boundary. However, by passing width and height messages between the outer and inner documents, our FRIV automatically resizes itself to fit its content. The parent page can override this behavior using stylesheets to specify a fixed or maximum size for the FRIV, or by explicitly intercepting the messages with its own handler. The page inside the FRIV must activate the automatic resizing feature explicitly by calling the `exportSize` function; this consent prevents an attacker web site from determining the number of bytes in a confidential document from another domain by measuring the FRIV's size.

The sugared FRIV also provides the illusion of direct communication via function calls; in the operating systems analogy, we provide an asynchronous remote procedure call abstraction. This abstraction retains the familiar syntax of contemporary mashups.

Suppose Alice hosts a mashup that uses third-party map software provided by Bob. Alice.com includes Bob's map software by rendering a FRIV that points to Bob.com. In order to allow Alice to recenter the map to a particular location, Bob defines a JavaScript method `setCenter(latitude, longitude)` and allows Alice to call it by using the syntax `parent.export(setCenter)`, where parent is the bottom-half FRIV at the root of Bob's DOM. Alice can now call Bob's `setCenter` method by invoking `exportedMethods.setCenter()` on the top-half FRIV that adjoins to Bob's SERVICEINSTANCE.

Alice can also export methods to Bob. For example, Alice might want to be notified when the user clicks on a point on the map. Alice can define the function `onclick(latitude, longitude)` and allow Bob to call it by calling the `export(onclick)` method of the top-half FRIV adjoining Bob's SERVICEINSTANCE. Bob can now call Alice's onclick method by invoking `parent.exportedMethods.onclick`.

The simple examples shown above do not involve

return values; we also provide a capability to make asychronous (using an additional JavaScript callback argument) calls to functions with return values. We also propose a synchronous interface, although synchronous RPC failure-couples the caller to the callee.

The FRIV tag is designed for web mashups where the caller and callee only partially trust each other. It is important to note that this syntactic sugar does not violate the data-only limitation of the communication channel (Section IV-C). If the participants trust one another enough (arguably completely) to pass code or references to DOM objects, then an inline `SCRIPT` tag suffices (in the operating systems analogy, the caller links to the third-party library directly).

One exception to the data-only rule is that a communication endpoint itself may be passed as an argument in a cross-SERVICEINSTANCE message; analogous in the OS world to passing a file descriptor through an IPC channel. This feature enables a gadget aggregator to directly connect two of its siblings. We have yet to solidify the details of this proposal.

VI. Proposed Implementation

For a proposal like MashupOS to be feasible, it must be implementable. We plan for future work a reference implementation of the MashupOS abstractions. Here, we consider alternative implementation approaches.

A. Isolation

Isolation of separate FRIVs, including commodity resource allocation, can be provided natively in the browser or a plugin, based on JavaScript type safety. Alternatively, isolation could be implemented using processes [10], virtual machines [1], or dynamic code transformation.

Code transformation offers an interesting alternative to the browser-upgrade deployment path. BrowserShield [11] is a framework that dynamically rewrites an HTML-and-JavaScript page to obey a given policy. One killer application of BrowserShield was to protect browser vulnerabilities by catching even dynamically-generated exploit code. A BrowserShield implementation of MashupOS would enforce isolation by preventing code from from following references across FRIV boundaries in the display DOM. Dynamic rewriting can incur significant performance overhead. Its advantage is that it can be deployed remotely, e.g. by a gadget aggregator site, to enhance legacy browsers with the MashupOS abstractions.

B. Incremental Deployment

Until all browsers support MashupOS, websites must handle users with legacy browsers that do not understand the FRIV tag, by providing a safe, user-friendly fallback behavior. To that end, we borrow a trick from IFRAME: FRIV-aware browsers shall ignore the contents of a FRIV tag. Legacy browsers ignore the unsupported FRIV tag, and proceed to render its contents normally. Thus, a web developer can place inside the FRIV tag a cross-domain IFRAME, ensuring that the referenced gadget is safely displayed, regardless of browser support for FRIV. Alternatively, the web developer may use the FRIV content to link to a FRIV-enabled browser upgrade or plugin.

To ensure that MashupOS web pages are considered valid XHTML, the FRIV tag is part of a custom XML namespace until it is someday adopted as part of the HTML standard. Thus, technically a FRIV should be created with the slightly longer syntax `<mashupos:friv src='page2.html' xmlns:mashupos= 'http://research.microsoft.com/mashupos/'`

Instead of introducing a new tag, an alternate approach would be to reuse an existing tag, such as IFRAME, SCRIPT, or OBJECT. Internet Explorer and Opera adopted this approach with the `security=restricted` attribute of IFRAME, which allows a containing page to render a frame with JavaScript and other active content disabled. Unlike the FRIV tag, the security-restricted IFRAME tag does not fail safely: When encountered by a browser such as Firefox that is not aware of the `security` property, the active content is allowed to execute.

VII. Related Work

Recently, a new wave of "web operating systems" [3] (e.g., YouOS [13]) have emerged. These sites present a traditional desktop user interface, complete with a window manager. The applications run natively in JavaScript. All are hosted on the same domain as the web desktop, and thus have unlimited access to one another. This lack of isolation, comparable to the 1995 PC desktop, requires the user to completely trust every application that is run.

Our earlier work on Subspace [7] provided a cross-domain communication mechanism that is designed to run on current browsers without any additional plug-ins or client-side changes. Subspace uses the browser's existing `document.domain` property to communicate with isolated subdomains, which are in turn used to draw in scripts from other domains. However, Subspace requires significant work on the part of the web developer to use correctly, and does not provide the resource constraint options of FRIV. We believe browsers should provide built-in cross-domain interaction primitives.

Crockford recently proposed the JSONRequest [2] communication mechanism for asynchronous cross-domain data retrieval from a remote server. The proposal

was motivated by scenarios where the cross-domain SCRIPT tag was being used to execute code when only data was really required. The JSONRequest's usage is similar to XMLHttpRequest, but it is not constrained by the Same-Origin Policy. Lifting these same-origin restrictions is safe because cookies are not sent, because the request includes a header indicating the source of the request, and because the server's reply must indicate the server is aware of the protocol and hence its security implications. JSONRequest transmits data in the JSON format, but its security applies equally to XML or other formats. A cross-domain JSONRequest is the client-side equivalent to a cross-server TCP request, and thus JSONRequest complements MashupOS well. Alternatively, one can simulate JSONRequest in MashupOS by creating a FRIV that communicates with its home domain and then passes the received data back to the outer SERVICEINSTANCE.

Crockford also identified the security limitations that affect today's cross-domain mashups. In response, he proposed a new HTML tag, the MODULE tag, to partition a page into a collection of modules [2]. A module groups DOM elements and scripts into an isolated environment; socket-like communications are allowed between the inner module and the outer module. To isolate the module from the origin server, modules may not make network requests. Thus, modules are equivalent to null-principal FRIVs.

Cross-document messages [4] are a proposed browser standard that would allow cross-domain frames to send string messages to each other on the client side. Cross-document messages are thus similar to the messaging capabilities of full-principal FRIVs. They are implemented in the Opera browser. We expect that with better data type support, automatic layout capabilities, and other syntactic sugar provided by FRIV, wider deployment and use of this messaging paradigm can be achieved.

VIII. CONCLUDING REMARKS

Client mashups enable a new generation of user-friendly and feature-rich web applications. While mashups turn the browser into a multi-user system with mutually distrusting domains as users, today's browsers offer web developers insufficient abstractions for integrating content from different domains: either cross-domain isolation with no communication or uncontrolled communication with no isolation. MashupOS applies operating system principles to bridge this gap. We introduce the SERVICEINSTANCE as the unit of resource isolation, data-only message-based communication between SERVICEINSTANCEs, and the FRIV as the abstraction of display sharing. Invoking well-understood operating system principles promises to provide a stable security foundation to replace today's teetering *de facto* abstractions.

REFERENCES

[1] R.S. Cox, J.G. Hansen, S.D. Gribble, and H.M. Levy. A Safety-Oriented Platform for Web Applications. In *Proc. IEEE Symposium on Security and Privacy*, 2006.

[2] D. Crockford. JSON. `http://www.json.org/`.

[3] Big WebOS roundup - 10 online operating systems reviewed. `http://franticindustries.com/blog/2006/12/21/big-webos-roundup-10-onlin%e-operating-systems-reviewed/`.

[4] Web Hypertext Application Technology Working Group. Web Applications 1.0, February 2007. `http://www.whatwg.org/specs/web-apps/current-work/`.

[5] Google Inc. Google Gadgets API Developer Guide. `http://www.google.com/apis/gadgets/fundamentals.html`.

[6] C. Jackson, A. Bortz, D. Boneh, and J. Mitchell. Protecting Browser State Against Web Privacy Attacks . In *Proc. WWW*, 2006.

[7] C. Jackson and H. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *Proc. WWW*, 2007.

[8] D. Kristol and L. Montulli. HTTP State Management Mechanism. IETF RFC 2109, February 1997.

[9] Windows Live Gadget Developer's Guide. `http://microsoftgadgets.com/livesdk/docs/default.htm`.

[10] C. Reis, B. Bershad, S. Gribble, and H. Levy. Using processes to improve the reliability of browserbased applications. In *Under submission*.

[11] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML . In *Proc. OSDI*, November 2006.

[12] J. Ruderman. JavaScript Security: Same Origin. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[13] YouOS. `http://www.youos.com/`.

Live Monitoring: Using Adaptive Instrumentation and Analysis to Debug and Maintain Web Applications

Emre Kıcıman and Helen J. Wang
Microsoft Research
{*emrek, helenw*}*@microsoft.com*

Abstract

AJAX-based web applications are enabling the next generation of rich, client-side web applications, but today's web application developers do not have the end-to-end visibility required to effectively build and maintain a reliable system. We argue that a new capability of the web application environment—the ability for a system to automatically create and serve different versions of an application to each user—can be exploited for adaptive, cross-user monitoring of the behavior of web applications on end-user desktops. In this paper, we propose a live monitoring framework for building a new class of development and maintenance techniques that use a continuous loop of automatic, adaptive application rewriting, observation and analysis. We outline two such adaptive techniques for localizing data corruption bugs and automatically placing function result caching. The live monitoring framework requires only minor changes to web application servers, no changes to application code and no modifications to existing browsers.

1 Introduction

Over the last several years, AJAX (Asynchronous JavaScript and XML) programming techniques have enabled a new generation of popular web-based applications, marking a paradigm shift in web service development and provisioning [11]. Unlike traditional web services, these new *web applications* combine the data preservation and integrity, storage capacity and computational power of data center(s) with a rich client-side experience, implemented as a JavaScript program shipped on-demand to users' web browsers[1]. This combination provides a compelling way to build new applications while moving the burden of managing an application's reliability from end-users to the application's own developers and operators.

Unfortunately, today's web application developers and operators do not have the end-to-end visibility they need to effectively build and maintain a dependable system. Unlike traditional web services, running exclusively in controlled, server-side environments, a web application depends on many components outside the developer's control, including the client-side JavaScript engine and libraries and the third-party back-end web services used by *mash-up* applications—web applications that combine functionality from multiple back-end web services. Of course, web application developers must also contend with the traditional bugs that occur when writing any large, complex piece of software, including logic errors, memory leaks and performance problems. When the inevitable problem does occur, the web application developer's lack of visibility into the heterogeneous client environments and the dynamic behavior of third-party services can make reproducing and debugging the problem practically impossible.

To address these challenges, we propose a *live monitoring* framework that exploits a new capability of the web application environment, *instant redeployability*: Each time any client runs a web application, the developers and operators of the application can automatically provide the client a new, different version of the application. Our live monitoring framework (1) exploits this capability to enable dynamic and adaptive instrumentation strategies; and (2) integrates the resultant on-line observations of an application's end-to-end behavior into the development and operations process.

Live monitoring enables a new class of techniques that use a continuous loop of automatic application rewriting, observation and analysis to improve the development and maintenance of web applications. Policy-based, automatic rewriting of application code provides the necessary visibility into end-to-end application behavior, and collecting observations on-line from live end-user desk-

[1] We make a distinction between a web service and a web application. The former includes only server-side components, while the latter also includes a significant client-side JavaScript component

Op	Performance (ms)	
	IE 7	**Firefox 1.5**
Array.join	35	120
+	5100	120

Table 1: The performance of two simple methods for concatenating 10k strings varies across browsers.

tops provides visibility into the real problems affecting clients. Distributing and sampling instrumentation across the many users of a web application provides a low-overhead instrumentation platform. Finally, using already-collected information to adapt instrumentation on-line enables efficient drill-down with specialized diagnosis techniques as problems occur.

2 Reliable Web Applications

The web application environment presents many of the same development and operations challenges that confront any cross-platform, distributed system. In this environment, however, there are also opportunities for a new approach to addressing these challenges.

Challenges

The root challenge to building and maintaining a reliable client-side web application is a lack of visibility into the end-to-end behavior of the program, brought about by the fact that execution of the web application is now split across multiple environments, including uncontrolled client-side and third-party environments and exacerbated by their heterogeneity and dynamics.

Non-standard Execution Environments: While the core JavaScript language is standardized as ECMAScript [7], most pieces of a JavaScript environment are not. The result is that applications have to frequently work-around subtle and not-so-subtle cross-browser incompatibilities. As a clear example, sending an XMLRPC request requires calling an ActiveX object in IE6, but a native JavaScript object in Firefox. More subtle are issues such as event propagation: *e.g.*, given multiple registered event handlers for a mouse click, in what order are they called? Moreover, even the standardized pieces of JavaScript can have implementation differences that cause serious performance problems (see Table 1 for examples of performance variance across browsers.)

Third-Party Dependencies: All web applications have dependencies on the reliability of back-end web services. While these back-end services strive to maintain high-availability, they can and do fail. Moreover, even regular updates, such as bug fixes and feature enhancements

App	JS (bytes)	JS (LOC)
Live Maps	1MB	54K
Google Maps	200KB	20K
HousingMaps	213KB	19K
Amazon Book Reader	390KB	16K
CNN.com	137KB	5K

Table 2: Numbers on the amount of client-side code in a few major web applications, measured in bytes and lines of code (LOC)

can break dependent applications. Anecdotally, such breaking upgrades do occur: Live.com updated their beta gadget API, breaking dependent developers code [13]; and, more recently, the popular social bookmark website, del.icio.us, moved the URLs pointing to some of their public data streams, breaking dependent applications [3].

Software Complexity: Of course, JavaScript also suffers from the traditional challenges of writing any non-trivial program[2]. While JavaScript programs were once only simple scripts containing a few lines of code, they have grown dramatically to the point where the client-side code of cutting-edge web applications easily exceed 10k lines, as shown in Table 2. The result is that web applications suffer from the same kinds of bugs as traditional programs, including memory leaks, logic bugs, race conditions, and performance problems.

The difficulties caused by heterogeneous execution environments and dynamic third-party behavior, as well as the challenge of writing correct software can certainly be improved through more complete standardization, better web service management and careful software engineering. But, we would argue that, at a minimum, software bugs and human error will continue to give all of these challenges a long life frustrating web application developers.

Opportunities

While the above challenges are faced by most any cross-platform distributed systems, two technical features of web applications provide an opportunity for building new kinds of tools to deal with these problems:

Instant Deployability: Web applications are deployed and updated by modifying the code stored on a central web server. Modulo caching policies, clients download a fresh copy of the application each time they run it, enabling instant deployability of updates. We take advan-

[2]Coding in JavaScript today is also made more difficult by a lack of compile-time errors and warnings, static type checking, and private scoping. We do not consider these problems fundamental, however, as current and upcoming tools, such as Google's WebToolkit are remedying these issues [8].

tage of this capability to serve different versions of a web application (*e.g.*, with varying instrumentation) over time and across users.

Dynamic extensions: During their execution, JavaScript-based web applications can dynamically load and run new scripts, allowing late-binding of functionality based on current requirements. We use this to download specialized fault diagnosis routines when a web application encounters a problem.

3 Live Monitoring

The goal of live monitoring techniques is to improve developer and operator visibility into the end-to-end behavior of web applications by enabling automatic, adaptive analysis of application behavior on real end-user desktops. At the core of a live monitoring technique is a simple process:

1. Use automatic program rewriting together with instant redeployability to serve differently instrumented versions of applications over time and across users.

2. Continually gather observations of the on-line, end-to-end behavior of applications running under real workload on many end-user's desktops.

3. As observations of application behavior are gathered and analyzed, use the results to guide the adaptation of the dynamic instrumentation policy.

4. In special cases, use the client's ability to dynamically load scripts to enable just-in-time fault diagnosis handlers, tailored based on previously gleaned information about the specific encountered symptoms.

Our framework for live monitoring, shown in Figure 1, divides this process across several key components. The **Transformer** is responsible for rewriting the JavaScript application as it is sent from the web application's servers to the end-user's desktop. The transformer contains both generic code, such as the JavaScript and HTML parsers reusable across many live monitoring techniques, and technique-specific rewriting rules. These rules are expressed in two steps: the first step searches for target code-points matching some rule-specific filter, such as "all function call expressions" or "all variable declarations"; and the second step applies an arbitrary transformation on a target code-point by modifying the abstract syntax tree of the JavaScript program. Each rewriting rule exposes a set of discrete knobs for controlling the rewriting of target code points. For example, a rule that adds performance profiling to function calls might expose an on/off knob for each function that could be profiled.

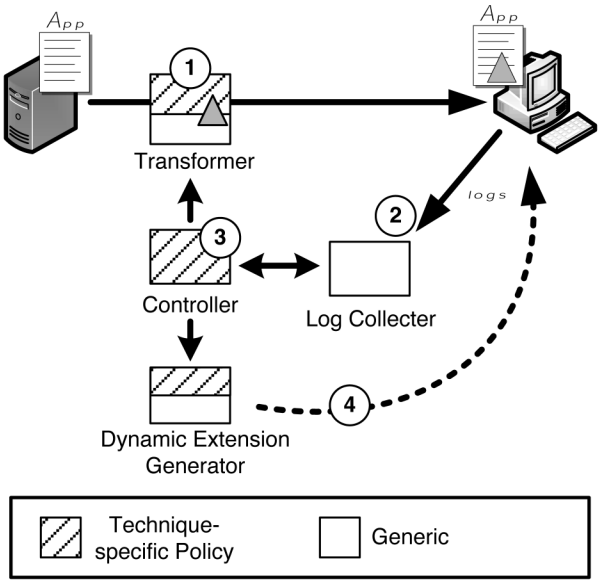


Figure 1: Live Monitoring Framework.

The **Controller** component is responsible for the core of the technique-specific adaptation algorithm, analyzing the collected observations of application behavior and using the results of the analysis to modify the knobs exposed by the rewriting rules in the Transformer. The **Log Collector** is a simple component, responsible for gathering observations returned by rewritten programs; and the **Dynamic Extension Generator** creates special-purpose fault diagnosis handlers, based on the application's request and configuration input from the Controller.

While some parts of this process are generic and reusable across techniques, the rest—what we call a live monitoring policy—is specific to each live monitoring technique. This policy includes the rewriting rules in the Transformer, the analysis policy in the Controller responsible for analyzing logs and modifying the knobs of the rewriting rules, and the dynamic extension generator.

4 Live Monitoring Policies

When developing a new policy to address a debugging or maintenance challenge, we consider several questions:

What are the appropriate rewriting rules? The first consideration when building a monitoring policy is what observations of application behavior need to be captured, and how a program can be modified to efficiently capture it. In particular, we ask what instrumentation is statically

written into the code, and what functionality will be dynamically determined and downloaded as needed from the Dynamic Extension Generator.

How does the rewriting adapt over time? A second consideration is which code points in a program should initially be rewritten, and how this choice changes over time as we gather more observations of behavior. The policy should also consider whether a multi-stage approach is appropriate, where completely different rewriting rules are applied to gather different kinds of information over time.

How does the policy spread instrumentation across users? A third axis of consideration is how a policy can distribute instrumentation across many users (*e.g.*, via sampling) and re-aggregate that information to reason about the program's overall behavior.

How do the developers and operators interact and use live monitoring policies? The final question when designing a policy is how people will use it. Some policies may be completely automated and continuously running, whereas other live monitoring policies may only run occasionally and on the explicit request of a developer. In particular, if the policy's application rewriting might affect the semantics of the program then human interaction is likely necessary.

We have built a prototype of our live monitoring framework, implemented several policies for debugging errors, drilling-down into performance problems, and analyzing runtime behavior to detect potentially correct cache optimizations and are exploring answers to these questions. The rest of this section describes two policies that use different styles of adaptation to address different problems. In the first example, a single rewriting rule is applied to different points in the code as we drill-down into data structure corruptions. The second example uses different rewriting rules over time, and decides where to place each rewriting rule based on observations gathered from previous application runs.

Locating Data Structure Corruption Bugs

While it can be very difficult to reproduce the steps to triggering bugs in a controlled, development environment, real users will run into the same problems again and again in a real, deployed application. We would like to capture the relevant error information and debug problems in real conditions, but adding all the necessary debugging infrastructure to the entire program can have too high an overhead. The solution is to adaptively enable the debugging infrastructure only when and where in the code it is needed.

Corruption of in-memory data structures is a clear sign of a bug in an application, and can easily lead to serious problems in the application's behavior. A straightforward method for detecting data structure inconsistencies is to use consistency checks at appropriate locations to ensure that data structures are not corrupt. A consistency check is a small piece of data-structure-specific code that tests for some invariant. *E.g.*, a doubly-linked list data structure might be inspected for unmatched forward and backward references. While today these checks are commonly written manually, there has been recent work on automatically inferring such checks [6].

When a consistency check fails, we might suspect that a bug exists somewhere in the executed code after the last successful consistency check[3]. If we execute these consistency checks infrequently, we will not have narrowed down the possible locations of a bug. On the other hand, if we execute these checks too frequently, we can easily cause a prohibitive performance overhead, as well as introduce false positives if we check a data structure while it is being modified.

Using live monitoring, we can build an adaptive policy that adds and removes consistency checks to balance the need for localizing data structures with the desire to avoid excessive overhead. Initially, the policy inserts consistency checks only at the beginning and end of stand-alone script blocks and event handlers (essentially, all the entry and exit points for the execution of a JavaScript application). Assuming that any data structure that is corrupted during the execution of a script block or event handler will remain corrupted at the end of the block's execution, we have a high confidence of detecting corruptions as they are caused by real workloads.

As these consistency checks notice data structure corruptions, the policy adds additional consistency checks in the suspect code path to "drill-down" and help localize the problem. As clients download and execute fresh copies of the application and run into the same data structure consistency problems, they will report in more detail on any problems they encounter in this suspect code path, and our adaptive policy can then drill-down again, as well as remove any checks that are now believed to be superfluous.

Several simple extensions can make this example policy more powerful. For example, performance overhead can be reduced at the expense of fidelity by randomly sampling data structure consistency across many clients. Also, if the policy finds a function that only intermittently corrupts a data structure, we can explore the program's state in more detail with an additional rewriting rule to capture the function's input arguments and other key state arguments and other state to help the developer narrow down the cause of a problem.

[3] JavaScript programs are executed within a single-thread, avoiding the possibility of a separate thread having corrupted the data structure.

Identifying Promising Cache Placements

Even simple features of web applications are often cut because of performance problems, and the poor performance of overly ambitious AJAX applications is one of the primary complaints of end-users. Some of the blame lies with JavaScript's nature as a scripting language not designed for building large applications: given a lack of access scoping and the ability to dynamically load arbitrary code, the scripting engine often cannot safely apply even simple optimizations, such as caching variable dereferences and in-lining functions.

With live monitoring, however, we can use a multi-stage instrumentation policy to detect possibly valid optimizations and evaluate the potential benefit of applying the optimization. Let us consider a simple optimization strategy: the insertion of function result caching. For this optimization strategy to be correct, the function being cached must (1) return a value that is deterministic given only the function inputs and (2) have no side-effects. We monitor the dynamic behavior of the application to cull functions that empirically do not meet the first criteria. Then, we rely on a human developer to understand the semantics of the remaining functions to double-check the remaining functions for determinism and side-effects. Finally, we use another stage of instrumentation to check whether the benefit of caching outweighs the cost.

The first stage of such a policy injects test predicates to help identify when function caching is valid. To accomplish this, the rewriting rule essentially inserts a cache, but continues to call the original function and check its return value against any previously cached results. If any client, across all the real workload of an application, reports that a cache value did not match the function's actual return value, we know that function is not safe for optimization and remove that code location from consideration. After gathering many observations over a sufficient variety and number of user workloads, we provide a list of potentially cache-able functions to the developer of the application and ask them to use their knowledge of the function's semantics to determine whether it might have any side-effects or unseen non-determinism. The advantage of this first stage of monitoring is that reviewing a potentially short list of possibly valid cache-able code points should be easier than inspecting all the functions for potential cache optimization.

In the second stage of our policy, we use automatic rewriting to cache the results of functions that the developer deemed to be free of side-effects. To test the cost and benefit of each function's caching, we distribute two versions of the application: one with the optimization and one without, where both versions have performance instrumentation added. Over time, we compare our observations of the two versions and determine when and where the optimization has benefit. For example, some might improve performance on one browser but not another. Other caches might have a benefit when network latency between the user and the central service is high, but not otherwise. Regardless, testing optimizations in the context of a real-world deployment, as opposed to testing only in a controlled pre-deployment environment, allows us to evaluate performance improvement while avoiding any potential systematic biases of test workloads or differences between real-world and test environments.

5 Related Work

Several previous projects have worked on improved monitoring techniques for web services and other distributed systems [2, 1], but to our knowledge, live monitoring is the first to extend developer's visibility into web application behavior on the end-user's desktop. Others, including Tucek *et al* [15], note that moving debugging capability to the end-user's desktop benefits from leveraging information easily available only at the moment of failure—we strongly agree. In [9] Liblit *et al* present an algorithm for isolating bugs in code by using randomly sampled predicates of program behavior from a large user base. We believe that the adaptive instrumentation of live monitoring can improve on such algorithms by enabling the use of active learning techniques [5] that use global information about encountered problems to dynamically control predicate sampling. Perhaps the closest in spirit to our work is ParaDyn [10], which uses dynamic, adaptive instrumentation to find performance bottlenecks in parallel computing applications.

6 Challenges and Implications

In summary, we have presented live monitoring, a framework for improving developers' end-to-end visibility into web application behavior through a continuous, adaptive loop of instrumentation, observation and analysis. As examples, we have shown how live monitoring can be used to localize bugs and analyze runtime behavior to detect and evaluate optimization opportunities.

We still face open challenges as we look to building a practical and deployable live monitoring system, such as the privacy issues of added instrumentation. While we believe that the browser's sandbox on web applications, together with the explicit trust users already give web services to store application-specific personal information (e-mails, purchasing habits, etc) greatly reduces the potential privacy concerns of extra instrumentation, there may be corner cases where live monitoring would pose a risk. Another challenge is to maintain the

predictability—predictable behavior and performance—of web applications as we dynamically adapt our instrumentation.

If successful, however, we believe the implications of instant redeployability may go beyond monitoring and also open the door to adaptive recovery techniques, including variations of failure-oblivious computing and Rx techniques [14, 12]. In these cases, the detection of a failure and the discovery of an appropriate mitigation technique in one user's execution of an application could be immediately applied to help other users, before they experience problems. At the moment, web applications are the most widespread platforms that have the capability for instantly redeployment. In the future, however, automatic update mechanisms and other centralized software management tools [4] might enable instant redeployability in broader domains.

References

[1] M.K. Aguilera, J.C. Mogul, J.L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of SOSP 2003*.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of OSDI 2004*.

[3] A. Bosworth. How To Provide a Web API. `http://www.sourcelabs.com/blogs/ajb/2006/08/how_to_provide_a_web_api.html`, Aug 2006.

[4] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M.S. Lam. The Collective: A Cache-Based System Management Architecture. In *Proceedings of NSDI 2005*.

[5] D.A. Cohn, Z. Ghahramani, and M.I. Jordan. Active Learning with Statistical Models. *Journal of Artificial Intelligence Research*, 4, 1996.

[6] B. Demsky, M.D. Ernst, P.J. Guo, S. McCamant, J.H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of ISSTA 2006*.

[7] ECMA. ECMAScript Language Specification 3rd Ed. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, Dec 1999.

[8] Google. Google web toolkit. `http://code.google.com/webtoolkit/`.

[9] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of PLDI 2005*.

[10] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11), Nov 1995.

[11] T. O'Reilly. What is Web 2.0. `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web20.html`, Sep 2005.

[12] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failure. In *Proceedings of SOSP 2005*.

[13] S. Rider. Recent changes that may break your gadgets. `http://microsoftgadgets.com/forums/1438/ShowPost.aspx`, Nov 2005.

[14] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and Jr. W.S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of OSDI 2004*.

[15] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Automatic On-line Failure Diagnosis at the End-User Site. In *Proceedings of HotDep 2006*.

End-to-end Web Application Security

Úlfar Erlingsson Benjamin Livshits Yinglian Xie

Microsoft Research

Abstract

Web applications are important, ubiquitous distributed systems whose current security relies primarily on server-side mechanisms. This paper makes the end-to-end argument that the client and server must collaborate to achieve security goals, to eliminate common security exploits, and to secure the emerging class of rich, cross-domain Web applications referred to as Web 2.0.

In order to support end-to-end security, Web clients must be enhanced. We introduce *Mutation-Event Transforms*: an easy-to-use client-side mechanism that can enforce even fine-grained, application-specific security policies, and whose implementation requires only straightforward changes to existing Web browsers. We give numerous examples of attractive, new security policies that demonstrate the advantages of end-to-end Web application security and of our proposed mechanism.

1 Introduction

Web applications provide end users with client access to server functionality through a set of Web pages. These pages often contain script code to be executed dynamically within the client Web browser.

Most Web applications aim to enforce simple, intuitive security policies, such as, for Web-based email, disallowing any scripts in untrusted email messages. Even so, Web applications are currently subject to a plethora of successful attacks, such as cross-site scripting, cookie theft, session riding, browser hijacking, and the recent self-propagating worms in Web-based email and social networking sites [2, 17, 24]. Indeed, according to surveys, security issues in Web applications are the most commonly reported vulnerabilities on the Internet [16].

The problems of Web application security are only becoming worse with the recent trends towards richer, "Web 2.0" applications. These applications enable new avenues of attacks by making use of complex, asynchronous client-side scripts, and by combining services across Web application domains [8]. However, the shift towards Web 2.0 also presents an opportunity for enhanced security enforcement, since new mechanisms are again being added to popular Web browsers.

Therefore, we believe it is time to rethink the fundamentals of Web application security. It is our position that the client Web browsers must be given a greater role in enforcing application security policies. In this paper, we support our position with examples and a simple end-to-end argument: constraints on client behavior are enforced most reliably at the client. We also propose *Mutation-Event Transforms*: a novel, flexible mechanism for client-side security policy enforcement.

1.1 Motivating Attacks

Of the current attacks on Web applications, those based on *script injection* are by far the most prominent. For example, script injection is used in cross-site scripting [1] and Web application worms [2, 24].

A script injection vulnerability may be present whenever a Web application includes data of uncertain origin in its Web pages; a third-party comment on a blog page is an example of such untrusted data. In a typical attack, malicious data with surreptitiously embedded scripts is included in requests to a benign Web application server; later, the server may include that data, and those scripts, in Web pages it returns to unsuspecting users. Since Web browsers execute scripts on a page with Web application authority, these returned scripts can give attackers control over the users' Web application activities [1, 22].

Script injection attacks typically affect non-malicious users and succeed without compromising Web application servers or networks. For example, in 2005, the self-propagating Samy worm on MySpace used script injection to infect over a million users [24]. As a MySpace user viewed the MySpace page of another, infected user, the worm script would execute and send a page update request to the server, causing the worm script to be included also on the viewing user's page.

In an attempt to prevent script injection, most Web application servers try to carefully filter out scripts from untrusted data. Unfortunately, such data sanitization is highly error prone (see Section 2.1). For example, the Samy worm evaded filtering, in part, by the unexpected placement of a newline character [24].

Script injection is just one means of attack: there are many ways to exploit Web applications by presenting them with attacker-chosen data. As we demonstrate in this paper, end-to-end Web application security is not only a reliable means to prevent these attacks. Our proposals for enhanced, client-side security enforcement also form a simple, flexible foundation for the general security of Web applications, including future, more complex Web 2.0 applications.

2 The Case for End-to-end Defenses

In general, it is often best to establish systems guarantees at the point where they are needed, with an end-to-end check, rather than with earlier, piecemeal checks [21].

This end-to-end argument applies directly to Web application security. Although security policies should be determined and specified at the server, enforcement of policies about Web client behavior should be guaranteed at the client. The corresponding server-side checks are difficult to perform and, in practice, incomplete in ways that enable attacks.

2.1 Server-side Defenses and their Limitations

Web applications must consider the possibility of malicious attackers that craft arbitrary messages, and counter this threat through server-side mechanisms.

However, to date, Web application development has focused only on methodologies and tools for server-side security enforcement (for instance, see [11, 13]). At most, non-malicious Web clients have been assumed to enforce a rudimentary "same origin" security policy [22]. Web clients are not even informed of simple Web application invariants, such as "no scripts in the email message portion of a page", since clients are not trusted to enforce security policies.

This focus on centralized server-side security mechanisms is shortsighted: server-side enforcement has difficulties constraining even simple client behavior. For example, to enforce "no scripts", the server must correctly model complex, dynamic client activities such as string manipulation, and take into account all possible client features and bugs. This entails server consideration of a myriad different tags, encodings, and operators for comments and quoting [20].

Server-side removal of scripts is especially difficult for Web applications that wish to allow visual formatting or other data richer than simple text. As shown below, there are many non-obvious means of causing code execution, including within formatting tags:

```
<SCRIPT/chaff>code</S\0CRIPT>
<IMG SRC=" &#14;  code">
<STYLE>li {list-style-image: url("code");}</STYLE>
<DIV STYLE="background-image:\0075\0072\006C...">
```

Furthermore, server-side enforcement is unsuitable for Web 2.0 cross-domain mashups [25], which may access third-party servers to load code and data. For instance, Web clients perform such access whenever a Web application embeds the Google Search AJAX API [5].

2.2 Client-side Defenses and their Benefits

As described above, many security policies are best enforced at the client. Web clients are the final authority on client behavior—including where script code is found, what that code is, and from where the code was loaded. If informed of Web application security policies by the server, properly enhanced clients could reliably enforce those policies.

```
HTMLDocument.prototype.__defineGetter__(
    "cookie",
    function(){ return null; }
);
```

Figure 1: A programmatic security policy that will reliably disallow all script access to document cookies in many existing Web browsers, if included at the top of pages returned by a Web application server [3].

At the same time, the majority of users are not malicious, and would enable client-side enforcement to avoid exploits such as cross-site scripting and Web-based worms. Even if only benign users with enhanced clients might perform security enforcement, those users would be protected, and all users would benefit from fewer attacks on the Web application.

Unfortunately, there are many obstacles to the adoption of new, enhanced security mechanisms in popular Web browsers. Even when such enhancements are practical and easy to implement, they may not be deployed widely. Therefore, to increase its chance of widespread adoption, a Web client security mechanism should be practical, simple, and flexible, and be able to enforce multiple, attractive policies on client behavior.

3 New Client-side Security Mechanisms

In this paper we propose enhancing Web clients with new security mechanisms that can not only prevent existing attacks, but are able to enforce all security policies based on monitoring client behavior. In particular, our new mechanisms support policies that range from disallowing use of certain Web client features (e.g., `IFRAME`s or `OBJECT`s) to fine-grained, application-specific invariants such as taint-based policies that regulate the flow of credit-card information input by the user.

Concretely, we propose that client-side enforcement proceed through a new client mechanism: *Mutation-Event Transforms*, or METs. METs are introduce here; some details like how to prevent their subversion are in Appendix A. METs allow Web application security policies to be specified at the server in a programmatic manner, such that those specifications can be used directly for enforcement at the client. In this, METs are similar to the code in Figure 1, and recent proposals such as BEEP [9].

In short, with METs, Web application servers specify security policies as JavaScript functions included at the top of pages returned by the server, and run before any other scripts. At runtime, and during initial loading, these MET functions are invoked by the client on each Web page modification to ensure the page always conforms to the security policy. Before a mutation takes effect, METs have the ability to transform that mutation, and the code and data of the page, which gives METs great flexibility in enforcement. In particular, METs can

be used to implement inlined reference monitors and edit automata for security-relevant client events, which allows METs to be used to specify and enforce any security policy based on monitoring client behavior [4, 26].

METs are both simple and straightforward to adopt: Web clients need only implement a single new primitive for mutation-event callbacks, and expose already-present events and data structures. Because policies are programmatic, they can readily account for browser variation and properly limit client-side enforcement on legacy Web clients (indeed, JavaScript code is already commonly used for compatibility and debugging purposes). Furthermore, security policy enforcement using METs requires only reasonable assumptions about the attacker.

3.1 Assumptions about the Attacker

METs can reliably defend against powerful attackers that are able to present Web clients with arbitrary code and data. In particular, the attacker may be modeled as an arbitrary, malicious script within Web application pages that are subject to MET enforcement.

The correctness of MET enforcement is of concern only to non-malicious users; it relies on network integrity and depends on assumptions about the server and clients. We trust that the Web application server has not been compromised, and properly includes METs at the top of returned Web pages; however, we assume that server code may have bugs such that the returned pages may contain arbitrary attacker-chosen data. We trust the Web clients to execute METs with proper semantics and to correctly enforce the fundamental same-origin policy [22]. Finally, we trust the programmatic security policies and that they correctly reflect the security goals of the Web application developers.

4 Policy Specification and Enforcement

Web application developers must have freedom in choosing security policies, and how they are derived. We propose specifying security policies using programmatic MET callback functions written in JavaScript. At runtime, these MET callback functions operate on each new (or updated) Web page and ensure that it conforms to the security policy, either through validation or transformation of the code or data within the Web page.

As we demonstrate in this section, METs have the appealing property that simple policies are easy to specify and enforce (much as in Figure 1). Even so, although Web application developers may guide security enforcement with Web page annotations, code for METs is likely to come as pre-packaged libraries, or be determined automatically at the server.

In particular, METs can be used for client-side enforcement of application-specific *dynamic security policies* determined automatically at the server from the natural constraints imposed by the structured composition of client pages (e.g., using frameworks such as ASP.NET AJAX [14] or GWT [6]). METs can also enforce other rich policies, such as those that apply to Web 2.0 cross-domain mashups [25], where application pages are composed outside the scope of server enforcement.

4.1 Examples of General, Basic Security Policies

On the following page, Figure 2 describes examples of general policies that apply to client Web pages, their script code, and the nodes and attributes of document data. On the same page, Figure 3 shows how these policies can be readily instantiated using MET callback functions; this code should be read in conjunction with Appendix A. In what follows, these policies are referred to by their number, in parentheses.

Policies (1), (3), and (6) are examples that restrict potentially dangerous types of document nodes, allow scripts only in certain portions of the document, or limit scripts to a whitelist of trusted scripts (as in [9]).

Policies (2), (4), and (5) validate the structure of certain data structures and scripts in Web pages, which can prevent many attacks (e.g., attacks that use malformed SQL queries [23]). Without such validation, malicious attacks may exploit may benign client-side code by presenting it with malformed data. For instance, without enforcement of policy (5), the Web client may at any time execute new, unexpected code where only a data return value was expected. (Client-side data validation may also reduce the number of round trips to the server.)

As shown in policies (7) through (9), the set of policies supported by METs are not restricted to actions that change the document structure of Web pages. METs can also support constraints on network access or access to security-critical client variables, such as the Web browser history, and enforce containment scopes between client-side gadgets and modules.

Finally, as demonstrated by (10), security policies based on METs may even include the code for a security-enhanced JavaScript interpreter, and ensure that it is used to execute all script code. Such a custom interpreter can implement dynamic taint propagation or other complex security policies.

For reasons of space, our example METs use several support routines, that would naturally be defined by security policy code. For example, the `matchURLDomain` function in (8) might match string URLs in a policy-specific manner, while the `outmostAttr` function in (9) might recursively walk up the document tree in order to find the attribute definition closest to the root. Similarly, policy-specific variables may encode security-relevant state such as in (1) for allowed ActiveX GUIDs (e.g., the Flash player), and in (2) the identity of a particular node in the document structure of a Web page.

(1) Disallow certain dangerous nodes or attributes. For instance, <IFRAME> nodes might be disallowed, and <OBJECT> nodes only permitted when instantiating the Flash player with known content.

(2) Data invariants on certain document subtrees. The Web page document is subject to invariants, even when modified dynamically at the client; e.g., blog comments must be a well-formed list of <DIV> nodes.

(3) Disallow scripts in certain parts of a Web page. A special case of (2), for instance to disallow use of <SCRIPT> nodes in untrusted blog comments.

(4) Scripts match valid, server-defined templates. An application of (2) to scripts: new, client-defined scripts may be allowed, but, for example, the `onHover` script code for a dynamically-inserted list item might be required to match `highlight(identifier)`.

(5) Cross-domain scripts return only data, properly. Instantiating (4) to prevent unexpected introduction of new code by cross-domain client-mashup applications: for instance, any script returned into a cross-domain <SCRIPT> node must have a syntax tree that matches `ajaxCallback(jsonDataValue)`.

(6) Limit scripts to a static, server-defined set. A static form of (4) that may simply match the hash of the script source text against a fixed "whitelist".

(7) Constrained access to object fields and methods. For instance, giving partial access to `document.cookie`, or limiting arguments to network-access methods.

(8) Proper network access via (cross-domain) URLs. URLs are subject to access control—both node-attribute URLs (e.g., on <IMG>) and the URLs used programmatically in scripts, (e.g., in an XML request).

(9) Containment of script activity to certain subtrees. Scripts can only modify certain document subtrees; thus, a gadget (or client-side mashup) for Web search might only be allowed to mutate a <DIV> for search results.

(10) Script execution by a secure interpreter. Scripts are not executed directly, but through a special, security-enhanced interpreter that may enforce (8), above, or even more fine-grained policies, such as variants of stack inspection or data tainting [4, 13].

Figure 2: A selection of attractive client-side security policies that can be readily enforced using programmatic MET callback functions. This list emphasizes general, widely applicable policies, while application-specific dynamic security policies are discussed further in the text (in particular in Section 4.2).

Type signature for MET callback functions

```
ExtendedNode
MET_callback(in Node script, // source of mutation
             in Node target, // target in Web page
             in ExtendedNode oldValue,
             in ExtendedNode newValue);
```

Example programmatic MET callback policies

(1) Limit `OBJECT` nodes (on `OBJECT` events):

```
var ok = (newValue.classid == theAllowedGUID);
return (ok) ? newValue : null;
```

(2) Data invariant on inserted `DIV`s (on `DIV` events):

```
if (target.id != insertionParentID) return null;
if (oldValue != null) return null;
var ok = ExtDOM.MatchStructure(newValue,
                "<div><ul><li></li></ul></div>");
return (ok) ? newValue : null;
```

(3) Limit script placement (on `SCRIPT` events):

```
var ok = ! findParentAttr("no_scripts", target);
return (ok) ? newValue : null;
```

(4) Restrict the code in scripts (on `SCRIPT` events):

```
var ok = ExtDOM.MatchScriptStructure(newValue,
                "highlight('identifier');"
return (ok) ? newValue : null;
```

(5) Proper data in AJAX replies (on `SCRIPT` events):

```
if (! newValue instanceof ScriptBody) return null;
var ok = ExtDOM.MatchScriptStructure(newValue,
                "callback({count: 1; sum: 5;});"
return (ok) ? newValue : null;
```

(6) Script whitelisting (on `SCRIPT` events):

```
var ok = whitelist[ hash(newValue.toString()) ];
return (ok) ? newValue : null;
```

(7) Disallow `history` access (on `SCRIPT` events):

```
if (! newValue instanceof ScriptBody) return null;
return ExtDOM.ReplaceScriptLiteral(newValue,
              "history", "fresh_unused_literal");
```

(8) Limit network access (on `SCRIPT` events):

```
var ok = matchURLDomain(newValue.src, "foo.com");
return (ok) ? newValue : null;
```

(9) Script containment (on any mutation event):

```
var src = outmostAttr("containment", script);
var dst = outmostAttr("containment", target);
return (src == dst) ? newValue : null;
```

(10) Secure interpreter (on `SCRIPT` events):

```
if (! newValue instanceof ScriptBody) return null;
var arg = ExtDOM.CreateScriptNode("Arguments",
                newValue.ToJSON());
var func = "special_js_interpreter";
return ExtDOM.CreateScriptNode("Call", func, arg);
```

Figure 3: The type signature of MET callback functions and several possible implementations for policies like those in Figure 2. The details in Appendix A are relevant to this code. Due to space constraints, only terse, uncommented code is shown and the functionality of policy-provided variables and methods is indicated by name.

Figure 4: An outline of the document tree for an aggregation Web page that contains both RSS news items and email messages.

4.2 Application-Specific, Dynamic Security Policies

Policies can also be highly application-specific. Such policies can be either hand-written by the application developer or generated through static analysis of the Web application. This is illustrated by the examples below.

Example 1. Access control within a page. Figure 4 shows an example of a DOM tree containing data from two RSS sources: `rss_1` and `rss_2`. We would like to make sure that `rss2_code` does not modify the first element so that it is impossible to have a rogue RSS feed that changes the contents of another one. Using policy (9), we can restrict code to modify only DOM elements declared within the same scope. This policy allows isolation of code and data on a single page, and refines the "same-origin" policy of existing Web clients.

Figure 4 also shows how security policy can be directed by inline attributes on document nodes. In this case, a `no_scripts` attribute is used to direct MET enforcement of a policy such as (3) in Section 4.1.

Example 2. Google Web Toolkit (GWT). In GWT, the developer writes his or her application in Java [6]; the application is subsequently compiled by the GWT into two parts: a Java part that resides on the server and a JavaScript part that resides on the client. Unfortunately, given a client-side attack, the assumptions of the original Java application may not hold for the scripts at the client,

To prevent this, the server may generate policies that enforce consistency properties of the client code. For example, the server may wish to ensure that access control properties such as "a private method may only be invoked by methods in the same Java class" present in the original Java source code are preserved in the JavaScript code on the client side. Instantiation of such a policy would be application-dependent and could be obtained through static analysis of the original Java code.

Example 3. Server-generated content templates. Dynamic policy generation is also relevant to ASP.NET or JSP pages. Both of these technologies allow servers to mix static HTML and dynamic content. Using static analysis (e.g., that in [15]), the computed parts of Web pages can be approximated and, thereby, the structure and contents of generated pages. For example, the analysis may be directed to assume no permitted scripts in application inputs. Such page "templates" are highly suitable for client-side enforcement.

5 Discussion

End-to-end Web application security entails preventing client behavior and server interaction that should be impossible, by construction, or has otherwise been determined to be illegal. Whether policies are driven by automatic analysis, or by manual setting of policy, there is much to gain from this form of security. In particular, it is a necessary foundation for securing Web 2.0 applications like cross-domain mashups, which are often outside the scope of existing mechanisms.

Mutation-event transforms, or METs, are an attractive option for client-side security. METs are flexible enough to enforce any security policy based on execution monitoring [4, 26]. In particular, METs readily allow precise enforcement of policies on both code and data (e.g., such as those in [23]). At the same time, METs, and their supporting code, should be straightforward to implement, since they rely only on existing browser events and data structures.

In comparison, the servers can leverage the "same origin" security policy [22] to enforce some client-side policies, as done in SessionSafe [10]. Such schemes require multiple, elaborate server domains that may be cumbersome to manage. Even so, they can provide only limited, coarse protection such as disallowing access to Web application cookies—as in policy (7) in Section 4.1.

Some previous proposals enforce client-side security policies by making use of separate proxies to rewrite server requests from the Web client. Noxes [12] places simple restrictions on the URLs of requests. BrowserShield [19] and CoreScript [26] use elaborate script rewriting techniques to enforce policies such as disallowing cookie access and dangerous tags—as in policies (1) and (7) in Section 4.1. Although they are useful (e.g., for legacy support), such proxy-based mechanisms must correctly parse data and code in requests, which can be a near-intractable problem, even using structured, formal methods (see Section 2.1 and [26, Section 6]).

Indeed, like METs, reliable mechanisms for client-side security policies must necessarily build on the final parsing of code and data performed at the Web client. This approach has been taken in previous mechanisms, most notably in BEEP [9], but also in [7]. However, these proposed mechanisms have provided little flexibility in security policy specification and enforcement, only supporting policies like (3), (6), and (7) in Section 4.1.

The enforcement of end-to-end security policies offers benefits to all Web application users, but requires changes to existing Web browsers. The inclusion of our proposed METs mechanisms in Web clients can reliably prevent existing attacks and provide a flexible, fine-grained foundation for the enforcement of future application-specific security policies

References

[1] CGI Security. The cross-site scripting FAQ. http://www.cgisecurity.net/articles/xss-faq.shtml.

[2] E. Chien. Malicious Yahooligans. http://www.symantec.com/avcenter/reference/malicious.yahooligans.pdf, 2006.

[3] S. Di Paola. Wisec security. http://www.wisec.it/sectou.php?id=44c7949f6de03, 2006.

[4] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proc. IEEE Security and Privacy*, 2000.

[5] Google AJAX search API. http://code.google.com/apis/ajaxsearch.

[6] Google Web toolkit. http://code.google.com/webtoolkit.

[7] O. Hallaraker and G. Vigna. Detecting malicious JavaScript code in Mozilla. In *Proc. IEEE Conf. on Engineering of Complex Computer Systems*, 2005.

[8] B. Hoffman. Ajax security. http://www.spidynamics.com/assets/documents/AJAXdangers.pdf, 2006.

[9] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, 2007.

[10] M. Johns. SessionSafe: Implementing XSS immune session handling. In *Proc. ESORICS*, 2006.

[11] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities. In *Proc. IEEE Symp. on Security and Privacy*, 2006.

[12] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *ACM Symp. on Applied Computing*, 2006.

[13] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proc. Usenix Security Symp.*, 2005.

[14] Microsoft ASP.NET AJAX. http://ajax.asp.net.

[15] Y. Minamide. Static approximation of dynamically generated Web pages. In *Proc. WWW*, 2005.

[16] MITRE. Common vulnerabilities and exposures. http://cve.mitre.org/cve/, 2007.

[17] Open Web Application Security Project. The ten most critical Web application security vulnerabilities. http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf, 2004.

[18] T. Pixley. DOM level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events, 2000.

[19] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proc. OSDI*, 2006.

[20] RSnake. XSS (Cross Site Scripting) cheat sheet. http://ha.ckers.org/xss.html, 2006.

[21] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[22] Same origin policy. http://en.wikipedia.org/wiki/Same_origin_policy, 2007.

[23] Z. Su and G. Wassermann. The essence of command injection attacks in Web applications. In *Proc. POPL*, 2006.

[24] The Samy worm. http://namb.la/popular.

[25] Web Mashup. http://www.webmashup.com.

[26] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *POPL*, 2007.

A Mutation-Event Transforms

Here, we present some details on METs, our proposed new mechanism for flexible client-side enforcement.

Mutation events are defined in the proposed Document Object Model (or *DOM*), level-2, as events caused by any action that modifies the document structure [18]. METs are similar to, but simpler than, these standards proposals. METs are also more expressive since they operate on extended data that include both the standard DOM tree model [18] and the abstract syntax trees (ASTs) of executable scripts. Both mutation events and the ASTs of scripts are abstractions already implemented in Web clients; thus, support for the METs primitive should not require substantial client additions or changes.

Importantly, METs provide two mutation events for <SCRIPT> nodes: first, an event when the script node is inserted in the DOM, and another event when that inserted node is populated with the AST for its script code. This separation allows METs to enforce security policies that limit network access caused by SRC host URL attributes in script nodes. Other nodes, such as <STYLE>, are also handled in this manner.

The type signature of MET callback functions is given at the start of Figure 3. Both the `script` and `target` are regular DOM nodes in the document tree. The `script` refers to the node containing the script that is attempting the document mutation (e.g., by writing to an `innerHTML` field). The `target` refers to the parent node where `newValue` is about to be inserted to replace `oldValue`. Both `oldValue` and the `newValue` are well-formed, properly nested subtrees of our extended DOM that includes ASTs; either may be `null` to denote empty subtrees. The callback functions return an extended DOM subtree to be used (instead of `newValue`).

MET callback functions may be registered for DOM elements of particular types, e.g., as follows :

```
add_MET_callback(nodeType, policy)
```

In the above, `policy` would be invoked whenever a DOM element of type `nodeType` is affected (i.e. inserted, replaced, or deleted). This would happen at runtime, right before the mutation, but after the Web client has parsed the new, proposed extended DOM values.

All programmatic security policy variables and functions, and MET callback registration, may naturally occur in the first <SCRIPT> tag of Web pages. To prevent subversion of security enforcement, the script language scoping rules (or other means) should prevent access to security policy code and further MET callback registration might be disabled, e.g., by simply setting `document.prototype.add_MET_callback` to `null` in the code. More flexibly, Web clients might allow other scripts to register MET callback functions, if they carefully ensure security policy always takes precedence.

HotComments: How to Make Program Comments More Useful?

Lin Tan, Ding Yuan and Yuanyuan Zhou
Department of Computer Science, University of Illinois at Urbana-Champaign
{*lintan2, dyuan3, yyzhou*}*@cs.uiuc.edu*

Abstract

Program comments have long been used as a common practice for improving inter-programmer communication and code readability, by explicitly specifying programmers' intentions and assumptions. Unfortunately, comments are not used to their maximum potential, as since most comments are written in natural language, it is very difficult to automatically analyze them. Furthermore, unlike source code, comments cannot be tested. As a result, incorrect or obsolete comments can mislead programmers and introduce new bugs later.

This position paper takes an initiative to investigate how to explore comments beyond their current usage. Specifically, we study the *feasibility* and *benefits* of automatically analyzing comments to detect software bugs and bad comments. Our feasibility and benefit analysis is conducted from three aspects using Linux as a demonstration case. First, we study comments' characteristics and found that a significant percentage of comments are about "hot topics" such as synchronization and memory allocation, indicating that the comment analysis may first focus on hot topics instead of trying to "understand" any arbitrary comments. Second, we conduct a preliminary analysis that uses heuristics (i.e. keyword searches) with the assistance of natural language processing techniques to extract information from lock-related comments and then check against source code for inconsistencies. Our preliminary method has found 12 new bugs in the latest version of Linux with 2 already confirmed by the Linux Kernel developers. Third, we examine several open source bug databases and find that bad or inconsistent comments have introduced bugs, indicating the importance of maintaining comments and detecting inconsistent comments.

1 Introduction

1.1 Motivation

Despite costly efforts to improve software-development methodologies, software bugs in deployed code continue to thrive, contributing to a significant percentage of system failures and security vulnerabilities. Many software bugs are caused by *miscommunication* among programmers, misunderstanding of software components, and careless programming. For example, one programmer who implements function `Foo()` may assume that the caller of `Foo` holds a lock or allocates a buffer before calling `Foo`. However, if such assumptions are not clearly specified, other programmers can easily violate them, introducing bugs.

The problem above is further worsened by software evolution and growth. Typically, industrial and open source software are written by numerous developers over long periods of time, e.g. more than 10 years, with programmers frequently joining and departing the software development process. As a result, miscommunication and misunderstanding become increasingly severe, significantly affecting software quality and productivity.

To address the problem, comments have been used as a standard practice in software development to increase the readability of code by expressing programmers' intentions in a more *direct, explicit, and easy-to-understand*, but less precise (i.e. ambiguous) way. Comments are written in natural language to explain code segments, to specify assumptions, to record reminders, etc., that are often not expressed explicitly in source code. For example, the function `do_acct_process()` in Linux Kernel 2.6.20 assumes that it is only called from `do_exit()`; otherwise it may lead to failure. Fortunately, this assumption is stated in the source code comments, so other programmers are less likely to violate this assumption. Similarly, the comment above function `reset_hardware()` states that the caller must hold the instance lock before calling `reset_hardware()`. Such comments are very common in software including Linux and Mozilla (as shown later in this paper).

Though comments contain valuable information, including programmers' assumptions and intentions, they are not used to their maximum potential. Even though they significantly increase software readability and improve communication among programmers, they have not been examined by compilers or program analysis tools, such as debugging tools. Almost all compilers and program analysis tools simply skip the comments and parse only the source code.

If compilers and static analysis tools could automatically extract information such as programmers' assumptions described above, the extracted information could be used to check source code for potential bugs. For example, if `do_acct_process()` is called from a func-

tion other than `do_exit()`, or if the instance lock is not acquired before calling `reset_hardware()`, it may indicate a bug. The compilers and static analysis tools could detect such bugs automatically by comparing the source code and the assumptions extracted from comments if they could automatically extract such assumptions.

While comments can help programmers understand source code and specify programmers' assumptions and intentions in an explicit way, **bad or obsolete comments** can negatively affect software quality by increasing the chance of misunderstanding among programmers. In practice, as software evolves, programmers often forget to keep comments up to date. These obsolete comments, no longer consistent with the source code, provide confusing, misleading and even incorrect information to other programmers, which can easily introduce new bugs later. Unlike source code that can be tested via various in-house testing tools, comments can not be tested by current tools. Therefore, if comments could be automatically analyzed and checked against source code for inconsistencies, such bad comments may be identified to avoid introducing new bugs.

Unfortunately, automatically extracting information from comments is very challenging [20] because comments are written in natural language, may not even be grammatically correct, and are a mixture of natural language phrases and program identifiers. Moreover, many phrases in software have different meanings from natural language. For example, the word "pointer" in software is associated with "memory" and "buffer". While natural language processing (NLP) techniques have made impressive progress over the years, they are still limited to certain basic functionalities and mostly focus on well written documents such as the Wall Street Journal or other rigorous news corpus. Therefore, to automatically *understand* comments, it would require combining NLP with other techniques such as program analysis, statistics, and even domain-specific heuristics.

1.2 Contributions

This position paper takes the first initiative to study the *feasibility* and *benefits* of automatically analyzing program comments to detect software bugs and bad comments. Our feasibility and benefit analysis is conducted from three aspects using Linux as a demonstration case:

- **Are there hot topics in comments?** To analyze comments, we first need to understand their characteristics. Our results show that, while different Linux modules have different hot topics, they also share common topics such as synchronization and memory allocation. Specifically, 1.2-12.0% of comments (in different modules) in Linux are related to locks, and a substantial percentage (3.8-17.0%) of comments are about memory allocation. These results indicate that instead of aiming for the prohibitively challenging task of understanding any arbitrary comments, we can focus our comment analysis on automatically extracting information related to only hot topics.
- **Are comments useful for detecting bugs?** Comments provide an *independent* and more explicit source of information compared to source code, and this information can be used to perform sanity (consistency) checks against source code for potential bugs or bad comments. As a proof of concept, we conduct a preliminary analysis that uses heuristics (i.e. keyword searches) with the assistance of basic natural language processing techniques to extract information from lock-related comments. We choose the lock topic as our demonstration case as it is one of the major topics in comments, and synchronization bugs can cause severe damage that is difficult to detect. In our preliminary results, comments helped us find 12 new bugs in Linux, with 2 confirmed by the Linux developers, demonstrating some promising results to motivate the research direction of automatic comment analysis.
- **Are bad comments causing harm in practice?** While it is conceivable that bad comments can mislead programmers, have they introduced new bugs in practice? Our preliminary bug reports examination finds that inconsistent comments did introduce new bugs in real world software such as Mozilla. This indicates that it is important for programmers to maintain comments and keep them up to date and consistent with code; and it is also highly desirable to automatically detect bad and inconsistent comments to avoid misleading programmers and introducing new bugs later. Moreover, we analyze several bug databases and find that at least 62 bug reports in FreeBSD [3] are about incorrect and confusing comments, implying that some programmers have already realized the harm that can be caused by bad comments.

2 Hot Topics of Comments

To find out whether program comments have "hot topics", we conduct a simple keyword frequency study on Linux's comments. In our analysis, *a comment* is defined as one comment sentence.

Table 1 shows the most frequently used keywords, *hot keywords*, in comments from five major Linux modules. As expected, many hot topics are module specific. For example, a substantial percentage of comments in the kernel modules contain keywords "signal", "thread", or "cpu", whereas many comments in the memory management module contain keywords "page" or "cache".

Interestingly, while different Linux modules have their own hot keywords, they share some common hot keywords such as "lock", "alloc" and "free". For example,

kernel			mm			arch			drivers			fs		
keyword	**%**	**Freq**	**keyword**	**%**	**Freq**	**keyword**	**%**	**Freq**	**keyword**	**%**	**Freq**	**keyword**	**%**	**Freq**
signal	8.1%	109	page	30.6%	331	bit	5.4%	2729	device	4.1%	6828	block	4.4%	3174
thread	7.7%	104	cache	11.2%	121	interrupt	5.2%	2650	data	3.7%	6129	inode	3.4%	2407
cpu	7.0%	94	map	10.3%	111	register	4.2%	2121	interrupt	3.6%	6093	file	2.9%	2069
process	6.8%	92	memory	9.8%	106	address	3.3%	1661	bit	3.1%	5260	buffer	2.3%	1680
kernel	6.8%	91	*alloc*	9.6%	104	copyright	3.2%	1616	register	3.1%	5163	page	2.1%	1470
lock	6.2%	83	***lock***	7.4%	80	pci	3.1%	1561	command	3.0%	5023	*alloc*	2.0%	1413
task	5.9%	79	*free*	7.4%	80	kernel	2.9%	1489	buffer	3.0%	4989	*free*	1.8%	1322
timer	4.7%	63	swap	6.8%	73	irq	2.9%	1452	driver	2.9%	4836	directori	1.5%	1108
check	4.5%	61	start	6.2%	67	...	...	...	...	...	...	***lock***	1.5%	1082
time	4.4%	59	mm	5.6%	60	***lock***	0.8%	384	***lock***	0.8%	1385	pointer	1.5%	1055

Table 1: Keyword frequency in Linux. % is calculated as the number of comments in a module containing the keyword over the total number of comments in the module. Freq is keyword frequency.

kernel		mm		arch		drivers		fs	
%	**Freq**	**%**	**Freq**	**%**	**Freq**	**%**	**Freq**	**%**	**Freq**
10.0%	135	12.0%	130	1.2%	600	1.2%	1983	2.3%	1667

Table 2: Lock-related keyword frequency in Linux. Noise such as "block" and"clock" is excluded.

0.8% to 7.4% of the comments in Linux, a total of 3014 comments, contain the word "lock". This is probably because often Linux code is reentrant and thereby requires locks to protect accesses to shared variables. As synchronization-related code is usually complicated and tricky with many assumptions, programmers commonly use comments to make the synchronization assumptions and intentions explicit.

Different keywords, e.g. lock, unlock, spinlock, and rwlock, are all about locks; however, they are considered separate keywords. Therefore, we improved our keyword rank techniques to find lock-related comments. We replace all lock-related keywords with "lock" and then count the total number of comments that contain "lock". The results are shown in Table 2. The percentage of comments that contain "lock" is then increased to 1.2-12.0%.

Similarly, keywords related to memory allocation and deallocation also appear in a significant portion of comments, 3.8% and 17.0% in the *fs* module and the *mm* module, respectively. This is because memory management is another important topic that requires developers to communicate with each other. Miscommunication can easily lead to memory related bugs, which can be exploited by malicious users to launch security attacks.

While so far we have studied comments only from Linux code, we believe that our results represent comments of most system software including operating system code and server code, because synchronization and memory allocation/deallocation are important yet error-prone and confusing issues for such software.

3 Comment Analysis

As a proof of concept, we conduct a preliminary study that combines natural language processing techniques and topic-specific heuristics to analyze synchronization-related comments in Linux and use the extracted information to detect comment-code inconsistencies. As the goal of this position paper is merely to motivate the research of automatic comment analysis by demonstrating its feasibility and potential benefits, the comment analysis in this paper is heuristic-based and cannot be used to extract comments of arbitrary topic—achieving such goal remains as our immediate future work.

3.1 Analysis Goals

As a feasibility study to demonstrate the benefit potential, the analysis in our preliminary study focuses on extracting lock-related programming rules. Specifically, the goal of our analysis is to extract lock-related information (referred to as "rules" in this paper) according to the eight templates listed in Table 3. These templates are designed based on our manual examination of comment samples from Linux. Some comments have positive forms such as "the lock must be held here", whereas some others are negative such as "the lock must not be held here". Therefore, the automatic comment analysis needs to differentiate negative and positive forms. Otherwise, it will badly mislead the sanity checks.

In addition to determining to which template a lock-related comment belongs, we need to find the specific parameter values, i.e. which lock is needed.

ID	Rule Template
1/2	L must (NOT) be held [for V] before entering F.
3/4	L must (NOT) be held [for V] before leaving F.
5/6	L_A must (NOT) be held [for V] before L_B.
7/8	L must (NOT) be held here.

Table 3: Example rule templates. Each row shows two rule templates, one positive and one negative. L denotes a lock. F is a function. V means a variable. Brackets ([]) denote optional parameters.

3.2 Analysis Process

To automatically understand what type of lock-related rule a comment contains is a challenging task. The reason is that the same rule can be expressed in many different ways. For example, the rule *"Lock L must be held before entering function F"* can be paraphrased in many ways, such as (selected from comments in Linux): (1) *"We need to acquire the write IRQ lock before calling ep_unlink()"*; (2) *"The queue lock with interrupts disabled must be held on entry to this function"*; (3) *"Caller must hold bond lock for write."* Therefore, to analyze comments, we need to handle various expressing forms.

A Simple Method. A simple method is to use some heuristics such as "grep" to search for certain keywords in comments. For example, we can first grep for comments that contain keyword "lock" to obtain all lock-related comments. We then look for action keywords "acquire", "hold", or "release" or their variants such as "acquired", "held" and "releasing". Afterward, we look for negation keywords such as "not", "n't", etc. to differentiate negative rules from positive ones.

While the method is simple and can narrow down the number of comments for manual examination, it is very inaccurate because it considers only the presence of a keyword, regardless where in the comment the keyword appears. The simple approach will make *mistakes* in at least the following three cases. First, if the action keyword is not in the main clause, the sentence may not contain an expected rule. For example, comment "returns -EBUSY if locked" from Linux does not specify a locking rule since "if locked" is a *condition* for the return value. Second, if the object of the action verb is not a lock, maybe no locking rule is contained. For example, a comment from Linux "lockd_up is waiting for us to startup, so will be holding a reference to this module, ..." contains "lock" and "hold", but the object of "hold" is not a lock, and no expected rule is contained. Third, a comment containing the keyword "not" does not necessarily imply the extracted rule is negative. For instance, "Lock L must be held before calling function F so that a data race will not occur", still expresses a positive rule.

Our Preliminary Method. To accurately analyze comments for lock-related rules, we extend the simple methods above with systematic natural language processing (NLP) techniques to analyze comment structures and word types.

We first break each comment into sentences, which is non-trivial as it involves correctly interpreting abbreviations, decimal points, etc. Moreover, unique to program comments is that sentences can have '*', '/' and '.' symbols embedded in one sentence. Furthermore, sometimes a sentence can end without any delimiter. Therefore, besides using the regular delimiters, '!', '?', and ';', we use '.' and spaces together as sentence delimiters instead of using '.' alone. Additionally, we consider an empty line and the end of a comment as the end of a sentence.

Next, we use a modified version of word splitters [7] to break a sentence into words. We then use Part-of-Speech (POS) tagging and Semantic Role Labeling techniques [7] to tell whether a word in a sentence is a verb, a noun, etc., to distinguish main clauses from sub clauses, and to tell subjects from objects.

Then we apply keyword searches on selected components of each comment. Specifically, we first search for keyword "lock" in the *main clause* to filter out those lock-unrelated comments. Then we check whether the keyword "lock" serves as the *object* of the verb or the *subject* in the main clause, and whether the *verb* of the main clause is "hold", "acquire", "release", or their variants. By applying these searches on the most relevant components, we can determine whether the comment contains a lock-related rule or not.

Finally, we determine the following information to generate the rule in one of the forms presented in Table 3.

Is the rule specific to a function? If we see words such as "call" or "enter function" in a sentence, then it is highly likely that the rule contained in the target comment is specific to a function associated with the comment (Template 1 - 4 in Table 3). In this case, we can automatically extract the function name from the source code. The intuition here is that a comment about a function is usually inserted at the beginning of the function. Therefore, a simple static analysis can easily find the name of the function defined right after the comment.

What is the lock name? The lock name of a rule is usually the object of the verb in the main clause, which is often explicitly stated in comments. Therefore, we can automatically extract it as our NLP tools can tell which word is the object.

Is the rule positive or negative? By identifying the verb and negation words, such as "not", we can determine whether the rule is positive (template 1, 3, 5, or 7) or negative (template 2, 4, 6, or 8). For example, a main clause containing verb "hold" without any negation word is likely to be positive, whereas a main clause containing verb "hold" with a negation word is likely to be negative.

Our analysis algorithm is still primitive and is now designed for lock-related comments, and we are in the process of improving its accuracy and flexibility to analyze comments of any topic selected by users.

3.3 Inconsistency Detection

After we extract the lock related rules, we scan the source code to detect comment-code inconsistencies. Our analysis is flow-sensitive and context-sensitive. For example, if a rule extracted from comments says that a lock L should be held before calling function F, our checker

```
drivers/scsi/in2000.c:
/*Caller must hold instance lock!*/
static int reset_hardware(…){…}
…
static int in2000_bus_reset(…){
   …
   reset_hardware(…);
   …
}
```

(a) The comment says that `reset_hardware()` must be called with the instance lock held, but no lock is acquired before calling it in the code.

```
drivers/pci/proc.c:
static void *pci_seq_start(…){
   …
   /*surely we need some locking
     *for traversing the list?*/
   while (n--)
      {p = p->next; …}
   …
}
```

(b) The comment states that a lock is needed when the list is traversed. But there is no lock acquisition in the code.

Figure 1: Two confirmed bug examples in Linux.

performs a static analysis from every root (without any caller, e.g. `main()`) of the call-graph to explore every path that may lead to function F to see if it acquires lock L before calling F. To improve efficiency, we first prune the control flow graph and call graph and only keep those nodes that are related to lock acquire, L, or function F. We also perform simple points-to analysis to handle variables that may be aliases of L. The details of our rule checkers are similar to [13, 16].

Although we use static checking to detect bugs, it is quite conceivable that rules extracted from comments can be checked dynamically by running the program.

3.4 Results: New Bugs Detected

We conducted a preliminary evaluation of the analysis on Linux, which automatically extracted 538 lock-related rules from five Linux modules shown below.

kernel	mm	arch	drivers	fs
29	16	50	263	180

We detected 12 bugs in Linux with 2 confirmed by developers by using 137 rules extracted from comments, and we are working on checking the rest of the rules. Figure 1(a) shows a confirmed bug in Linux. The comment above `reset_hardware()` states that the caller must hold the instance lock, but the lock is not acquired when the function is called from `in2000_bus_reset()`, which can cause data races. This bug was fixed in Linux by adding `spin_lock_irqsave(instance->host_lock, flags)`. As shown in Figure 1(b), a comment says a lock is needed to traverse the list. However, no lock is used for accessing the list in the code. This bug was fixed by adding proper locking for accessing the list.

4 Bad Comments

If we can automatically detect bad comments, we can help prevent these bad comments from confusing and misleading programmers, who consequently introduce bugs. In this feasibility study, we manually study bug reports from several Bugzilla databases to find out if bad comments have caused new bugs.

```
nsCRT.h:
//must use delete[] to free memory
//allocated by PR_strdup
static PRUnichar* PR_strdup(…);
…
nsComponentManager.cpp:
buf = PR_strdup();
…
delete [] buf;
```

```
Quote from Bug Report 172131
in Mozilla Bugzilla:

"nsCRT.h's comment suggests the
wrong De-allocator.

nsComponentManager.cpp actually
uses the wrong De-allocator."
```

Figure 2: Bad comments that caused a new bug in Mozilla. Code is modified to simplify illustration.

Such a real world bug example from Mozilla (Revision 1.213 of nsComponentManager.cpp) is shown in Figure 2. This bug was introduced because the programmer read and followed an incorrect comment, as indicated by the description in the Bugzilla bug report: *"nsCRT.h's comment suggests the wrong De-allocator. nsComponentManager.cpp actually uses the wrong De-allocator"*. Misled by the incorrect comment, "must use `delete[]` to free the memory", a programmer used `delete[]` to free the memory pointed by `buf`, resulting in a bug as reported to Mozilla's Bugzilla database [6]. In a later version (Revision 1.214 of nsComponentManager.cpp), this bug was fixed by replacing `delete[] buf` with `PR_free(buf)`. The incorrect comment has also been fixed accordingly (in file nsCRT.h).

Moreover, we found that at least 62 bug reports in FreeBSD [3] are about incorrect and confusing comments, indicating that some programmers have realized the importance of keeping comments updated.

5 Related Work

Extracting rules and invariants from source code. Many bug detection tools [10, 11, 16] have been proposed to extract rules or invariants from source code or execution traces to detect bugs. Unlike these tools, our study automatically extracts programming rules from *comments*. Our approach also allows the detection of bad comments that can introduce bugs later.

Empirical study of comments. Woodfield, Dunsmore and Shen [19] conducted a user study on forty-eight experienced programmers and showed that code with comments is likely to be better understood by programmers. Jiang and Hassan [15] studied the trend of the percentage of commented functions in PostgreSQL. Recent work from Ying, Wright and Abrams [20] shows that comments are very challenging to analyze automatically because they have ambiguous context and scope. None of these propose any solution to automatically analyze comments or detect comment-code inconsistencies.

Annotation language. Annotation languages [4, 9, 12, 14, 21] are proposed for developers to comment source code using a formal language to specify special information such as type safety [21]. Previous work titled "comment analysis" [14] automatically detects bugs caused

by wrong assumptions made by programmers. However, what they refer to as "comments" are essentially annotations written in a formal annotation language, not comments written in natural language that are used in most existing software and are analyzed in our work.

While these annotation languages can be easily analyzed by a compiler, they have their own limitations. First, these annotation languages are not as expressive or flexible as natural language, often only expressing simple assumptions such as buffer lengths and data types. Additionally, they are not widely adopted because developers are usually reluctant to learn a new language. Finally, millions of lines of comments written in natural language already exist in legacy code. Due to all these reasons, our approach well compliments the annotation language approach since we analyze general comments written in natural language. Rules inferred by our approach from comments can also be used to *automatically* annotate programs to reduce manual effort.

Automatic document generation from comments. Many comment style specification tools are proposed and are widely used to automatically build documentation from comments [1, 2, 5, 8]. Since these specification tools restrict only the format but still allows programmers to use natural language for the content (i.e. they are semi-structured like web pages), automatically "understanding" or analyzing these comments still suffers from similar challenges to analyzing unstructured comments.

Comment and document analysis for software reuse. Matwin and Ahmad [18] used natural language processing techniques to extract noun phrases from program comments in LINPACK (a linear algebra package) to build a function database so that programmers can search the database to find routines for software reuse. Another study [17] built a code library by applying information retrieval techniques on documents and comments. But none of these work attempts to "understand" the information contained in comments to automatically checked against code for inconsistencies.

6 Conclusions and Future Work

In this paper, we study the feasibility and benefits of automatically analyzing comments to detect software bugs and bad comments. Our preliminary results with real world bugs and bad comment examples have demonstrated the benefits of such new research initiative. We are in the process of continuing exploring this idea in several ways. First, we are improving the accuracy and *generality* of our comment analysis algorithm. Second, we are applying our algorithm to extract other types of rules such as memory-related rules, to detect other types of bugs, and to detect bad comments. Third, we are studying the characteristics of comments from other software to validate that our observations from Linux comments are representative. So far, our examinations of Mozilla and Apache have shown results similar to Linux.

7 Acknowledgments

We thank the anonymous reviewers for their useful feedback and the Opera group for useful discussions and paper proofreading. This research is supported by NSF CNS-0347854 (career award), NSF CCR-0325603 grant, DOE DE-FG02-05ER25688, and Intel gift grant.

References

[1] C# XML comments. http://msdn.microsoft.com/msdnmag/issues/02/06/XMLC/.

[2] Doxygen. http://www.stack.nl/~dimitri/doxygen/.

[3] FreeBSD problem report database. http://www.freebsd.org/support/bugreports.html.

[4] Java annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[5] Javadoc tool. http://java.sun.com/j2se/javadoc/.

[6] Mozilla Bugzilla database. https://bugzilla.mozilla.org/.

[7] NLP tools. http://l2r.cs.uiuc.edu/~cogcomp/tools.php.

[8] RDoc. http://rdoc.sourceforge.net/.

[9] SAL annotations. http://msdn2.microsoft.com/en-us/library/ms235402.aspx.

[10] D. R. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP '01*.

[11] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE'00*.

[12] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 2002.

[13] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A system and language for building system-specific, static analyses. In *PLDI'02*.

[14] W. E. Howden. Comments analysis and programming errors. *IEEE Trans. Softw. Eng.*, 1990.

[15] Z. Jiang and A. Hassan. Examining the evolution of code comments in PostgreSQL. In *MSR '06*.

[16] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE'05*.

[17] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng.*, 1991.

[18] S. Matwin and A. Ahmad. Reuse of modular software with automated comment analysis. In *ICSM '94*.

[19] S. Woodfield, H. Dunsmore, and V. Shen. The effect of modularization and comments on program comprehension. In *ICSE'81*.

[20] A. Ying, J. Wright, and S. Abrams. Source code that talks: An exploration of Eclipse task comments and their implication to repository mining. In *MSR'05*.

[21] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI'06*.

Towards a Practical, Verified Kernel

Kevin Elphinstone* Gerwin Klein* Philip Derrin Timothy Roscoe†
Gernot Heiser*‡
National ICT Australia§

Abstract

In the paper we examine one of the issues in designing, specifying, implementing and formally verifying a small operating system kernel — how to provide a productive and iterative development methodology for both operating system developers and formal methods practitioners.

We espouse the use of functional programming languages as a medium for prototyping that is readily amenable to formalisation with a low barrier to entry for kernel developers, and report early experience in the process of designing and building seL4: a new, practical, and formally verified microkernel.

1 Introduction

We describe our approach to constructing seL4 — a useful yet formally verified operating system kernel, by means of a novel development process which aims to reconcile the conflicting methodologies of kernel developers and formal methods practitioners.

Despite vigorous debate on the topic of microkernels versus virtual machine monitors [5, 6, 12], there is an emerging consensus on smaller and more trustworthy kernels (whether hypervisors or microkernels) at the core of larger software systems. We have argued that the small size of current kernels, and the increased power of interactive theorem proving environments, means that the time is right to attempt formal verification by proof of a real-world microkernel [14].

The end goal of such a project is to show that a working kernel implementation behaves as it is formally specified in an abstract model. Additionally, we would like properties such as spatial partitioning of processes to hold in both the model and implementation, together with useful properties such as guaranteed termination of system calls, and the kernel never throwing an internal exception.

Successful OS kernels have generally been the result of careful attention to performance issues, and repeatedly iterating bottom-up implementations of low-level functionality, in some cases changing high-level interfaces and functionality to accommodate implementation constraints and performance goals. This is, unfortunately, in conflict with formal methods, which typically work by top-down refining models of system properties, and rarely deal with low-level implementation features.

This paper describes our approach to resolving this tension, and reports on our experience so far in applying it to seL4. We use a high-level language (Literate Haskell) to simultaneously develop a specification of the kernel and a reference implementation for evaluation and testing. The implementation can be used in conjunction with a simulator such as QEMU for running real application binaries, while the specification generates input to an interactive theorem prover (Isabelle) for formal proof of properties. The use of a clean, high-level language allows rapid iterative prototyping of both the specification and reference implementation. Finally, a deployable kernel is constructed as a refinement of the reference implementation in a high-performance low-level language.

The rest of this paper is structured as follows. In the next section we look in more detail at the issues in achieving a verified kernel, based in part on our experience trying to formally verify L4. Section 3 describes our pragmatic approach to tackling the issues identified, and Section 4 reports on our experience so far with seL4. Section 5 concludes.

2 Background and Issues

There are many challenges in designing, specifying, implementing, and formally verifying a high-performance microkernel. In our view, the most significant of these (and our focus in this paper) is reconciling the approach taken by kernel developers when system building with that taken by formal methods practitioners in developing

*Also at the University of New South Wales
†Now at ETH Zürich, Switzerland
‡Also with Open Kernel Labs
§National ICT Australia is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

and verifying properties of a system.

Kernel developers tend to adopt a bottom-up approach. Required functionality is provided by iteratively developing a high-performance low-level implementation, and it is not unusual to modify the delivered functionality or its interface to facilitate efficient implementation.

In contrast, formal methods practitioners take a top-down approach, iteratively developing potential models of the system to possess the properties required, with secondary regard (if any) to low-level implementation details.

This characterization simplifies a rather complex problem, but it illustrates the need for a methodology that has a low barrier to entry for both teams, facilitates both working together, and enables both to efficiently iterate through the design, specification, implementation, and verification of the system.

Creating an assured and useful general-purpose OS kernel has been a goal for some time [1, 16]. Recently, a number of approaches have been adopted.

A strawman approach is to create a natural-language specification and then iterate through the design of the system. Such a specification is easily written and read, but is prone to ambiguity and incompleteness. It often fails to expose design issues that may have a significant impact on performance, usability, and ease of implementation.

The VFiasco project [7] aims to verify an existing kernel (L4/Fiasco) directly by developing a formal semantics for the subset of C++ used to build it, in particular with a novel treatment of memory access. However, a formal semantics for a sufficiently rich subset of C++ is a large task, and it is unclear how much progress has been made since the project's inception in 2001.

The Coyotos team [13] take the different approach of defining a new low-level implementation language (BitC) with precise formal semantics, and hope to subsequently verify properties of the kernel they are building.

Although with less emphasis on high-level verification, the Singularity project also uses a type-safe imperative language (C#), but with additional compiler extensions to allow programmers and system architects to specify low-level checkable properties of the code, for example IPC contracts [3].

All these approaches iteratively develop a kernel in an imperative systems programming language (with varying degrees of safety), and then attempt to reason at a some level about the system as a whole. The challenge here is that it may be extremely difficult to extract an abstract model from the finished artifact, as the expected behavior is not made clear by the low-level code (especially since this code may contain bugs).

Furthermore, since it must be extracted from the implementation, such an abstract model cannot be used during the design process and is unlikely to be useful as a readable specification for developing a formal model of the system.

A final, and rarely acknowledged drawback with a bottom-up approach to verified kernel development is that many low-level details such as hardware interfacing must be implemented before any experience can be gained with the new design. The approach in section 3 allows a new design to be tried with real applications at an early stage.

In contrast, using formal specification at an abstract level to specify the design avoids ambiguity, but may not expose issues affecting performance and ease of implementation of the design until a much later stage. This is a particular problem for systems software, which is performance-critical and must operate in a relatively constrained environment. To a formal model, it makes little difference if a data type is implementable in four or five bytes, but to a kernel developer this can be critical to performance of an important code path in the system.

Also, it is difficult to evaluate the usability of a microkernel interface for building complete systems based on that interface, until such a system has actually been built.

Finally, the tools and techniques used for developing formal specifications are quite different to those typically used for systems software, so there is a high cost of entry for many kernel developers.

Implementation in a high-level language with well-defined and safe semantics is a good compromise between the previous two approaches. For example, the Osker kernel [4] is written in Haskell. The resulting implementation is easier to reason about than one in a low-level language but is typically limited by a high-level language's dependency on a complex runtime ill-suited to use in a standalone kernel. This may impose restrictions on the system that are not present when using low-level languages, such as a need for garbage collection of kernel memory.

In summary, there is a need for a development methodology that enables kernel developers to rapidly iterate through prototype kernels with sufficient access to low-level details to explore performance aspects of the design, while providing formal verification teams with the precise semantics of the system in a form suitable as input to a theorem proving environment.

We now describe our approach, which has produced a precisely specified kernel API, together with a usable reference implementation, and a formal model for the implementation in the Isabelle theorem prover.

3 Our Approach

In this section we describe the pragmatic approach we took to address the issues we identified earlier and unify our team of formal verification experts with our team of kernel developers. Referring to Figure 1, our approach revolves around "running the manual": We use Literate

Haskell to develop both a specification document of the kernel, and at the same time, a reference implementation that can be used for evaluation and testing. The Haskell specification serves as the medium for iterative prototyping of the implementation as well as the system model for both the kernel and formal modelling teams, i.e. the Haskell specification forms a bridge between the teams improving the flow of ideas, with a low barrier of entry for both. In addition, the reference implementation, when coupled with a simulator, can be used to run native binaries.

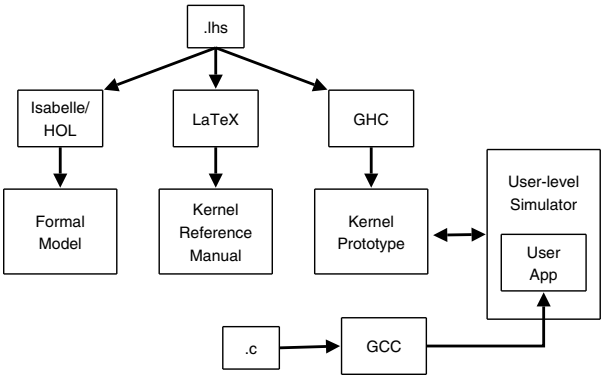


Figure 1: Graphical representation of our approach using Literate Haskell (.lhs) as a basis for specification, implementation, and formalisation

3.1 Kernel Development

From the kernel development perspective, various designs and their implementation can be explored at a high level without the initial need to deal with the complexity of low-level hardware. However, given that *the specification is an implementation*, kernel developers are forced to think about implementation details that would be necessary for efficient implementation on real hardware. While the Haskell implementation is not suitable for quantifying the kernel's performance, it does provide valuable insights into the approximate performance of data structures and algorithms.

To explore the utility of the design from a user-level perspective, we have several approaches. From the kernel perspective, the hardware is an event generator (interrupts, exceptions, system calls). The Haskell prototype is set up as the recipient of an event stream, upon which it can process the events and return the results as if it were a real kernel. Early, simplistic, versions of the kernel used a simple event generator function which took embedded pseudo-assembly to exercise the kernel model. For more mature versions of the design, we coupled the kernel model with a simulator for the unprivileged part of a real processor's ISA. This enables running compiled native code just as on real hardware. We currently can link our kernel model with the M5 Alpha simulator [8], a home-grown ARM simulator, and the QEMU ARM simulator complete with emulated devices. In each case, the kernel model processes the incoming event stream, returning the results such that it appears to application code that it is running on raw hardware. Thus we have an environment that allows kernel developers to explore design and implementation of both the kernel itself and the applications intended to be supported.

3.2 Formal Modelling

One of the tasks of the formal verification team is to extract a formal model of the prototype in order to reason about it in the theorem proving environment.

Given the precise semantics of the Haskell language, and the lack of side-effects of functional languages in general, it is a much simpler task to extract a formal model of the kernel compared to typical low-level systems languages like *C*.

The translation from Haskell to a model in the theorem prover Isabelle/HOL [11] is mostly syntactic and can be automated. The exceptions worth noting are lazy evaluation and monadic computations (an example being computation that modifies global state). While Isabelle/HOL is not suitable for expressing the semantics of lazy evaluation as provided by Haskell, our goal is not to translate faithfully every language construct in Haskell to Isabelle. Instead, we only seek an accurate representation of the semantics of each function that occurs in the prototype, and thus we can avoid the issue by not making essential use of laziness in our Haskell specification. The type system of Isabelle/HOL is also not strong enough to express monads in the traditional abstract way, but it can express all the particular concrete monads that are used in the prototype. For more detailed coverage of the issues we encountered in the translation process, see [2].

Since Isabelle/HOL is a logic of total functions, we had to prove during the translation that all functions terminate. The translation of our Haskell kernel model into Isabelle thus already establishes one useful property of the kernel — system calls always terminate.

In our ongoing work on formally verifying the kernel we are currently showing that the Isabelle/HOL translation of the Haskell prototype conforms to a simplified, more abstract formal model of the kernel. This model is used to facilitate proofs of more complex safety and invariant properties of the kernel without going into implementation detail.

The process of formal refinement already requires us to show certain invariants of the kernel. The main part of these invariants resemble a strong typing system: capabilities always point to kernel objects of the right type

(i.e. a thread capability always points to a valid TCB), capability tables are always of the correct size, references in kernel objects point to valid other kernel objects of the right type, etc. Note that the usual programming-language type systems are not strong enough to ensure these properties statically, even Haskell's very strong type system is insufficient.

Isabelle is an interactive theorem prover. This means that proof scripts are written manually with considerable creative input. The tool mechanically checks the proofs and assists in finding them by dealing with symbolic calculations, automatated proof tactics for certain classes of formulae etc, but it is not fully automatic.

The abstract specification is ca. 3.5k lines of Isabelle code, the translated Haskell prototype comes to about 7k lines of Isabelle code (this number is somewhat inflated due to the automated translation process), and the proof scripts to date to about 48k lines. The verification process so far lead to 109 changes in the abstract specification and 37 changes in the Haskell code. This supports the conclusion that executing the specification finds many small problems with relatively little effort early in the process.

Examples of the bugs we found range from cut & paste errors (e.g. using the wrong function on the `AsyncEndpoint` data type where the line directly above has the same pattern for `Endpoint`), over forgotten cases, to more conceptual issues like a complex, recursive delete function that was misbehaving in the case of circular pointer structures, or simply functions that were less general than believed and required more checks on user-supplied parameters (e.g. a capability move function that took the same arguments as the corresponding copy function, but would lead to security violations in some of the cases that worked for copy).

The next step in the verification will be connecting this prototype with a high performance *C* implementation of the seL4 API. Tuch et al [15] have demonstrated the technology for this step and have shown its feasibility for low-level *C* code in a case study on the L4 kernel memory allocator.

3.3 Overall

It should be clear that our approach makes some progress towards resolving the issues we have identified, but what might not be clear is how our approach relates to our original goal of producing a formally verified, high-performance microkernel — i.e. a kernel implemented in a more traditional systems language such as *C*.

Figure 2 illustrates the end game. We are using the mature Haskell specification as a basis for both a formal abstract model of the system, and a high-performance *C* implementation. To achieve our original goal, we expect to then show that the *C* implementation is a refinement of

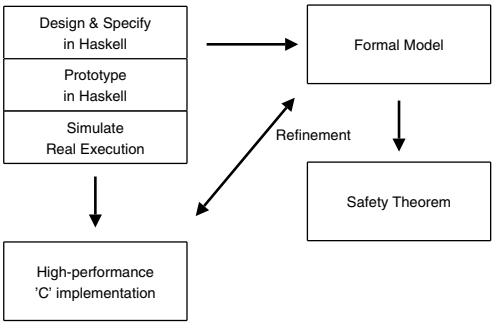


Figure 2: Overall approach to eventual verification of a high performance kernel.

the formal model. Details on our method for reasoning abstractly about low-level *C* code can be found in [15]. Together with the proof described here, this automatically gives us a proof that the abstract invariants also hold for the *C* implementation, and that the production kernel formally implements the abstract seL4 API as described previously [14]. As in the first verification step, we expect this second refinement to lead to a number of changes in the implementation — be they for performance or correctness reasons. For the final theorem to hold, these will be propagated back to the higher specifation levels and the proofs adjusted accordingly. Since the proofs are machine checked, we still get guaranteed consistency between all layers.

In principle, for the production kernel and its formal proof, the Haskell prototype could be thrown away; in the correctness sense it is redundant. For investigating new features and further developing the API, we expect it to be still useful, though, even when the production kernel exists. In any case, the Haskell kernel has already had an immense impact on overall productivity.

4 Experience

Despite the inevitable culture clash, experience with developing an OS kernel in this way has so far been positive. We describe our key learnings to date below.

4.1 Parallel Development

For us the most positive outcome of developing a kernel in a functional language has been having a medium in common for both kernel developers and formal modellers to cooperatively and iteratively develop a formally verified design and implementation of a small kernel.

The translation to Isabelle/HOL started relatively early, when the seL4 API was nearing a first stable point and first user-level binaries could be run through the machine simulator. The formal verification team, in translating the

Haskell specification, found and fixed a number of problems. An illustrative example is an obscure corner case, where the execution time of the IPC send operation was unbounded. This was discovered when Isabelle demanded termination proofs for operations that were supposed to execute in constant time.

This shows that formalisation and the use of theorem proving tools is beneficial even if full verification is not yet performed. Thus far, the cost involved in formalisation has been significantly less than the design, implementation, and testing input by the kernel team, while the kernel team did not have to switch to completely new methods or notations. Additionally, the common medium has enabled the formal modellers to have input on the structure of the reference implementation in order to reduce the complexity of formalisation, with minimal effect on the kernel behaviour and performance.

The user-level simulation environment has enabled the porting of existing software to the new kernel design prior to its existence on bare metal. The experience gained by actual use of the new design has also led to the identification of issues requiring attention. For example, when attempting to implement a higher-level system upon the microkernel, we found that an atomic swap operation on a particular kernel object greatly simplified the implementation of higher-level system software. The missing operation was added in a matter of hours, and formalised soon afterwards.

Summarising, we have found our methodology has enabled the kernel developers, the formal modellers, and the higher-level system programmers to work more closely together, leading to faster and better results than we would expect if the phases had been sequential.

4.2 Precise Specification

Our choice of Literate Haskell as our modelling language has enabled us to produce a reference manual and implementation that is one and the same thing, providing rare but highly-welcome assurance that our reference manual and reference implementation are consistent. Our catch phrase is "we run the manual". While our hope is to produce a readily understandable reference manual describing each operation with the reference Haskell implementation as the definitive definition of each operation, structuring our code to avoid too much implementation detail (that would obscure the relevant details of the specification) has proved challenging. However, the document is improving with each iteration.

4.3 Hardware and Prototyping

We found that iteratively prototyping the system in a high-level language away from the pitfalls and traps of real hardware helped in maturing the design of a new system. Rather than spend time debugging low-level code from the beginning of prototyping, we could initially focus on design and implementation issues of the basic concepts behind the system. As the design evolves, we are bringing in hardware-related issues (such as dealing with pages table or TLBs) when we choose to tackle each particular aspect of the design.

However, we could still gain experience in using the new design as soon as it was mature enough to be coupled with various user-level simulators. We have ported the Iguana OS (an embedded OS personality for the L4 microkernel [10]) to our design and could understand the interaction between Iguana and our new design prior to any prototype existing on bare metal.

5 Conclusions

We found that using a very high-level language as a medium for concurrently prototyping the specification and design of a high-performance microkernel not only provided a convenient and highly productive fast prototyping environment. More importantly, it allowed us to *design a high-performance kernel for formal verification*, producing a model that can be translated automatically into the theorem prover, and that is suitable for proving system invariants as well as formal refinement. Specifically it provided the bridge that makes it feasible, even easy, for kernel developers and formal methods people to collaborate on the specification, design, implementation and formal verification of the kernel.

Overall, this has allowed us to take a new approach towards building an OS kernel that can be proven to operate correctly. Almost forty years ago, Needham and Hartley remarked [9]:

> In designing an operating system one needs both theoretical insight and horse sense. Without the former, one designs an ad hoc mess; without the latter one designs an elephant in best Carrara marble (white, perfect, and immobile).

We believe that we have developed an approach to OS design that results in a highly productive synthesis of theoretical insight and horse sense.

References

[1] William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[2] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *ACM SIGPLAN Haskell WS*, Portland, OR, USA, Sep 2006.

[3] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proc. of EuroSys2006*, April 2006.

[4] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *Proc. 10th ACM Int. Conf. on Functional Programming*, 2005.

[5] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kottsovinos, and Dan Magenheimer. Are virtual machine monitors microkernels done right? In *10th HotOS*, Sante Fe, NM, USA, Jun 2005. USENIX.

[6] Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. Are virtual-machine monitors microkernels done right? *Operat. Syst. Rev.*, 40(1):95–99, Jan 2006.

[7] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *Proc. 2nd ECOOP Workshop on Programm Languages and Operating Systems*, Glasgow, UK, Oct 2005.

[8] The M5 simulator system. `http://m5.eecs.umich.edu/`, 2006.

[9] R. M. Needham and D. F. Hartley. Theory and practice in operating system design. In *2nd SOSP*, 1969.

[10] Iguana. `http://www.ertos.nicta.com.au/iguana/`, 2007.

[11] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[12] Timothy Roscoe, Kevin Elphinstone, and Gernot Heiser. Hype and virtue. In *11th HotOS*, San Diego, CA, USA, May 2007.

[13] Jonathan Shapiro. Coyotos. `www.coyotos.org`, 2006.

[14] Harvey Tuch, Gerwin Klein, and Gernot Heiser. OS verification — now! In *10th HotOS*, pages 7–12, Santa Fe, NM, USA, Jun 2005. USENIX.

[15] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *34th POPL*, pages 97–108, Nice, France, Jan 2007.

[16] Bruce Walker, Richard Kemmerer, and Gerald Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.

Beyond Bug-Finding: Sound Program Analysis for Linux

Zachary Anderson,[1] Eric Brewer,[1] Jeremy Condit,[1] Robert Ennals,[2]
David Gay,[2] Matthew Harren,[1] George C. Necula,[1] Feng Zhou[1]

[1] *University of California, Berkeley*
{zra,brewer,jcondit,matth,necula,zf}@cs.berkeley.edu

[2] *Intel Research Berkeley*
{robert.ennals,david.e.gay}@intel.com

Abstract

It is time for us to focus on *sound analyses* for our critical systems software—that is, we must focus on analyses that ensure the *absence* of defects of particular known types, rather than best-effort bug-finding tools. This paper presents three sample analyses for Linux that are aimed at eliminating bugs relating to type safety, deallocation, and blocking. These analyses rely on lightweight programmer annotations and run-time checks in order to make them practical and scalable. Sound analyses of this sort can check a wide variety of properties and will ultimately yield more reliable code than bug-finding alone.

1 Introduction

The strength of the systems community has long been *optimization*, historically for performance. This "quantitative approach" promotes common metrics and benchmarks for key properties that in turn become metrics of success for new work. But for properties such as security or privacy or safety, mere optimization is not sufficient. For such properties we need *guarantees*, and we believe that in many cases, it is feasible for the programmer and the compiler to provide these guarantees. For example, type checkers for strong type systems routinely prove deep facts about large programs without imposing an unnecessary burden on the programmer or the compiler. Guarantees for many important higher-level system properties can be obtained by similarly practical techniques, even for existing systems.

In the realm of static analysis for systems code, the systems community's focus on optimization has been manifested as a focus on *bug-finding*, where the metric of performance is the number of bugs found. Although this work has been very successful at revealing specific flaws in existing software, what we really want is a guarantee that no bugs of a specific type can occur—in other words, we want *sound static analysis*.

The systems community has historically felt that soundness requires too much programmer effort to be practical, either because it requires the programmer to write complex proofs, or because it requires large-scale rewriting of software in higher-level languages. In this paper, we argue that it *is* practical to provide many important soundness properties for large-scale systems software, even when written in C, and indeed that this is the approach that the community should be taking.

To demonstrate the feasibility of this approach, we present three soundness tools that we have developed for use with the Linux kernel. First, *Deputy* checks that a pointer always points to valid data of the correct type, even in presence of pointer arithmetic. Second, *CCount* checks that objects are only freed when there are no dangling references to them. Finally, *BlockStop* checks that the kernel does not call blocking functions while interrupts are disabled.

These tools have several properties that we believe are essential to making soundness practical for large-scale systems software such as the Linux kernel:

- **Lightweight, untrusted annotations:** Some annotations to existing source code may be required, but they are minimal, and they extend the type declarations to express simple ideas that should make sense to normal programmers. These annotations are not trusted by the compiler, so errors in the annotations will be caught along with errors in the code.

- **Incremental porting:** It is not necessary to annotate an entire program at once in order to gain any benefit. The system can be made safe one file or even one line at a time, with increasing levels of safety as more code is annotated.

- **Hybrid checking:** Most operations are checked statically, and the rest are checked at run time. Although detecting bugs at compile time is preferable, run-time checks are often necessary for practicality.

- **Erasure semantics:** The tools check, but do not otherwise modify, the behavior of a program. Annotations are written such that they can be ignored ("erased") by the traditional build process. The program is thus not locked into the tool.

- **Trusted code:** Sometimes the behavior of a particular code fragment is too complex for a practical tool to be able to guarantee soundness. In this case, the programmer should be allowed to mark the code as trusted for the purpose of analysis, thus raising its priority for code reviews and testing.

In addition, since we have written and inferred many annotations in the course of this work, we propose a shared repository of annotations and properties inferred for the Linux kernel. This repository, discussed in Section 3, would allow researchers to better collaborate when building such sound analyses in the future.

The major contribution of this paper is the idea that sound static analysis is a feasible and desirable alternative to bug-finding. In support of this idea, we present three analyses that we have used successfully on a working version of the Linux kernel, showing that it is possible to apply sound static analysis tools at a large scale. In addition, we present the basic principles of these tools that have allowed us to achieve this scalability.

Of course, bug-finding tools still have their place in the systems community. Heuristically checking complex properties of systems is often much easier than designing a sound static analysis, and in some sense, it can be viewed as a precursor to a sound analysis. However, we urge the community to focus on such sound analyses whenever possible, since guarantees provide more lasting benefits than bug-finding alone.

In the following section, we discuss each of our three analyses in more detail. Then, we discuss some future directions for research in sound analyses. Finally, we discuss related work and conclude.

2 Sound Analyses for Linux

In this section, we discuss three analyses we have applied to the Linux kernel thus far. This work was performed on a stripped-down version of the Linux 2.6.15.5 kernel, which consists of 435,000 lines of code. This code includes the basic kernel (`kernel/`, `mm/`, and `arch/i386/`), the networking stack (`ipv4`), several file systems (e.g., `ext2` and `procfs`), and several device drivers. The resulting kernel is fully functional in VMware as well as on several machines that the authors use. More details about our experimental kernel are available at `http://ivy.cs.berkeley.edu/`.

We focused on this stripped-down kernel in order to get a working system in place as fast as possible; however, with sufficient manpower, there is no reason to believe that these results could not be extended to the full Linux distribution. In other words, we omitted parts of the Linux kernel for manpower reasons, not for technical reasons.

2.1 Type Safety: Deputy

One major source of errors in Linux is the lack of type safety in C programs. Although a significant portion of C code is type safe, there are a number of language features and programming idioms whose safety cannot be verified by the C compiler. For example, verifying the correctness of array indices, union field references, and type casts is the sole responsibility of the programmer.

Our approach to handling this problem in the Linux kernel is to use the Deputy type system [CHA+07]. Deputy allows programmers to annotate pointer types with bounds information written in terms of other variables in the environment. Deputy also allows annotations for unions, null-terminated sequences, and polymorphic data. In return, Deputy is able to enforce the memory and type safety of the program using a combination of static checking and run-time checks. Note that these annotations are *not* trusted by the compiler, so if the programmer introduces an erroneous annotation, that error will be caught along with any errors in the code itself. In cases where Deputy's type system is insufficient to annotate the code properly, Deputy allows the programmer to explicitly mark code that should be trusted by our tools.

Overall, Deputy guarantees that, at run time, the value of every program expression corresponds to its compile-time type, and in doing so, Deputy prevents out-of-bounds array accesses and misuse of unions. Deputy assumes that trusted code is correct and that code outside the current module conforms to the provided annotations. Since all safety guarantees are relative to these assumptions about trusted or external code, we attempt to use Deputy on as much of the source code as possible.

Unlike other safe C variants such as Cyclone [JMG+02] and CCured [NCH+05], Deputy is incremental and thread safe. That is, programmers are free to add annotations and modify code function-by-function. This is possible because Deputy does not change the representation of the data visible across function boundaries, which allows "deputized" modules to interoperate with standard modules. While the initial version of the file may contain several blocks of trusted code, subsequent versions will gradually eliminate this trusted code in favor of fully annotated and checked code. The same holds of run-time checks: programmers can gradually modify the code to reduce the number of checks that must be

deferred until run time. This approach provides an incremental path towards a fully-annotated and type-safe Linux kernel.

In order to convert code to use Deputy, we replace `gcc` with `deputy` in the kernel makefiles. When Deputy is invoked on a C source file, it prints errors for any code that is considered illegal in its type system, such as casts between pointers with different base types. In order to resolve such errors, the programmer must add annotations (such as information about bounds or polymorphism), alter the code, or tell Deputy to trust the code. Once Deputy accepts the code, it will insert any necessary runtime checks and then compile with `gcc`. Booting the new code typically results in a number of warning messages due to incorrect or incomplete annotations; once these are revised, Linux runs as expected.

In previous work [ZCA+06], we described our experience using Deputy on Linux device drivers alone; however, we have now applied Deputy to the full 435,000-line kernel described above. We added standard Deputy annotations to approximately 2627 lines (about 0.6%), and we trusted approximately 3273 lines (less than 0.8%). (Note, however, that the significance of trusting a given line of code varies greatly according to how each line of code is used in the program text and how often it is executed at run time.) This conversion required approximately 7 person-weeks, but the conversion speed increased significantly in the later part of the process due to improvements in the tool and improvements in our ability to identify and apply common annotation patterns.

The `dep` columns of Table 2.1 show the *relative* performance of a 1.6 GHz Pentium M system with the Deputy-enabled kernel compared to the original Linux kernel, measured with the `hbench` [BS97] suite of benchmarks. Benchmarks in the `bw_` column are bandwidth tests (larger is better) and those in the `lat_` column are latency tests (smaller is better). Most tests show that Deputy incurs relatively small overhead. The worst cases are a maximum slowdown of 17% for the local TCP bandwidth test, and a 48% of latency increase for the local UDP latency test. We believe that these results are promising, since they suggest that type safety can be achieved at a modest performance cost.

bw_...	Rel. Perf.		lat_...	Rel. Perf.	
tests	dep	mem	tests	dep	mem
bzero	1.01	1.00	connect	1.10	2.10
file_rd	0.98	0.99	ctx	1.15	1.17
mem_cp	1.00	1.00	ctx2	1.35	1.13
mem_rd	1.00	1.00	fs	1.35	2.73
mem_wr	1.06	0.99	fslayer	1.04	0.98
mmap_rd	0.85	0.93	mmap	1.41	1.21
pipe	0.98	0.97	pipe	1.14	1.12
tcp	0.83	0.66	proc	1.29	1.30
			rpc	1.37	2.01
			sig	1.31	1.23
			syscall	0.74	1.22
			tcp	1.41	2.55
			udp	1.48	1.82

Table 1: Relative performance of the Linux kernels modified for Deputy (dep) and CCount (mem) on the `hbench` benchmark suite. Networking results measured with localhost.

2.2 Deallocation: CCount

Memory management bugs are a significant cause of software failures and vulnerabilities in C programs. If an object is freed when references to it still exist, then subsequent accesses to the freed object may actually access a new object that has been allocated in the same space, resulting in crashes, security vulnerabilities, and violations of type-safety properties assumed by other analyses.

The standard way to avoid memory management problems is to use garbage collection, where objects are automatically freed when they are no longer referenced. However, while previous work shows that it is possible to build an operating system kernel that uses garbage collection for memory management [HLA+05, Mit96, WP97], we believe that retrofitting a garbage collector onto a large legacy kernel such as Linux would be extremely difficult since it would require making significant changes to the way the kernel manages memory.

To address this issue, we have designed CCount, a C-to-C compiler and runtime system that uses reference counting to check the correctness of a C program's existing manual memory management. CCount's compiler modifies all pointer writes to maintain an 8-bit reference count on each 16-byte chunk of memory (a 6.25% space overhead), and the runtime system uses this data to check that frees are safe. Bad frees of objects with $k*256$ references will be missed by such a system, but we expect this case to occur very infrequently in non-malicious code. For total safety, an overflow check could be used.

Using CCount for the Linux kernel required two significant changes. First, we modified Linux's memory management routines to check reference counts and to zero all allocated storage (necessary to avoid decrementing random reference counts when initializing pointers).[1] On failure, we log an error and (optionally) leak the object to guarantee soundness. Second, CCount rewrites pointer writes such as '`*a = b`' to '`RC(b)++, RC(*a)--, *a = b`', where `RC` accesses the reference count of a pointer.[2] To support concurrent code, we must increment and decrement reference counts using atomic operations, and we must ensure that the increment happens before the decrement to avoid

transitory zero reference counts. In contrast, we assume that all pointer writes are already protected by appropriate locks and so we do not translate the write itself into an atomic operation. In the future, we plan to check that this assumption does indeed hold by using an additional static analysis tool.

CCount requires accurate type information when objects are freed, copied (`memcpy`), or cleared (`memset`). This information is generally similar to information needed by Deputy, so we expect to reuse Deputy annotations in the future. However, we currently have to provide some of this information "by hand". On our small kernel, we had to describe the layout of 32 types, use explicit runtime type information in 27 places, and change 50 uses of `memset` and `memcpy` to type-aware versions. We believe that such modifications are acceptable as long as they are made directly by the programmer at the source level, where the programmer can consider their consequences for performance and correctness, as opposed to being made automatically by the compiler.

With these changes, CCount will boot and run our small Linux kernel, but it reports many bad frees. We fix these bad frees by setting breakpoints at the bad free report statement and tracking down the cause of the bad free using our debugging facilities. Fixes to bad frees involve nulling out some extra pointers, usually around the time the corresponding object is freed (27 instances so far[3]) and adding *delayed free scopes* (26 so far). A delayed free scope simply delays all frees (and the associated reference count check) that happen inside it until its end, greatly simplifying the checks for complex or cyclical data structures. We have spent approximately 6 person-weeks porting CCount and making these changes to the Linux kernel, and we can now verify the correctness of all of the $\sim$107k frees that occur from boot time until the login prompt is available. And after running the `hbench` benchmark suite, our modified kernel reports 677 "bad" frees from $\sim$100M free calls. We are confident that we can eliminate these remaining bad frees.

The relative performance of the CCount-kernel vs the original Linux kernel is shown in the `mem` columns of Table 2.1. These numbers are measured on a 2.33GHz Intel® Xeon® 5140, running a uniprocessor kernel. The performance degradation is reasonable ($< 30\%$) except on the networking and file system tests. Additionally, we estimated the performance impact of CCount with an SMP kernel by changing the increment and decrement operations for reference count updates to use "locked" instructions. This leads to significant degradations in performance: `bw_tcp` decreases to 0.50, `lat_tcp` increases to 3.60, `lat_ctx` increases to 1.81, etc. Clearly these results need improving, which we hope to achieve through a combination of manual tuning, compiler optimization and a better runtime system.

2.3 Call Graph Analysis: BlockStop

There are many other invariants beyond type and memory safety that must be enforced in the kernel. One tool that is useful for several of these invariants is a call graph. Once we know which functions can be called where, we can begin to analyze important control-flow properties.

BlockStop is a whole-program analysis to enforce the requirement that the kernel does not call any functions that may block while interrupts are disabled, such as while holding a spinlock or handling an interrupt. Once we've run this analysis, we can emit an annotation for each function (and function pointer) that might eventually call a blocking function. Not only is this information useful for humans trying to understand the code, but the annotations can be checked incrementally whenever a file is changed, which preserves separate compilation.

A call graph is a directed graph where each node corresponds to a function and each outgoing edge represents the functions that it might call. The major challenge is to account for calls through function pointers. We use a whole-program *points-to analysis* to determine which functions a given pointer could refer to. Thanks to the type safety provided by Deputy and CCount, this points-to analysis is sound, except that we do not currently detect function calls made within inline assembly.

To find which functions might block, we annotate certain functions with a new `blocking` attribute, such as `copy_to/from_user`, `wait_for_completion`, etc. Allocators such as `kmalloc` have a special annotation to denote the fact that they may block if they are called with the `GFP_WAIT` flag. We then propagate this information backwards through the call graph to get a sound approximation of the set of functions that might block.

We ran this analysis on our test kernel and found one bug. We also encountered false positives, mostly due to the overly-conservative points-to analysis of function pointers. Replacing our simple points-to analysis with one that is field- and context- sensitive would improve the results, but to resolve these false positives quickly, we turned to run-time checks. We defined a special function that panics if interrupts are disabled, and we manually inserted calls to this function in 15 places in the kernel. For example, `read_chan` is a blocking function that BlockStop's points-to analysis incorrectly believes can be called by `flush_to_ldisk` while interrupts are disabled. Adding this run-time check to the start of `read_chan` reflects our assertion that this function will not actually be called by `flush_to_ldisk`. By ruling out these infeasible control-flow paths via run-time checks, we allow BlockStop to verify the desired property for our experimental kernel without any warnings or false positives.

3 Looking Forward

We believe that the three previous analyses represent only the tip of the iceberg in terms of sound analyses that can be used effectively on systems such as Linux. Here we discuss proposals for future analyses as well as ideas for making the results of these analyses widely available.

3.1 Future Analyses

There are many other opportunities to create sound analyses with the properties we discussed.

First, we are in the early stages of designing a hybrid checking tool for verifying *lock safety* in Linux. In addition to checking that deadlocks are impossible by verifying that the code uses a consistent locking order, this analysis will check Linux-specific invariants such as the requirement that the same spinlock is not acquired in interrupts and in process context with interrupts turned on. Light annotations will be used to name the locks, and run-time checks will be used when static checking does not suffice. We rely upon type and memory safety guarantees provided by Deputy and CCount.

Second, the call graph built for BlockStop can be used to prevent *stack overflow*. Given a sound call graph and information about the size of each stack frame, as in the Capriccio thread package [BCZ+03], we can ensure that every possible chain of function calls stays within its allotted 4 or 8 kB of stack space. Stack space annotations on each function will enable incremental verification. For recursive calls, run-time checks will be needed.

As a third example, it is possible to create a simple analysis for ensuring that *error codes* are properly checked at call sites. Programmers can annotate each functions with the set of codes that the function could return, or the programmer could simply indicate to the compiler that negative constant return values are error codes. Then a flow-sensitive analysis at call sites could verify that each of the error codes are accounted for, either together or separately. Calls through function pointers could use a merged list of codes from the functions that the pointer may alias.

Further examples include user/kernel pointers, tainted data flow, and concurrency issues such as identifying shared and thread-local data. All of these properties can be checked by analyses that follow the framework outlined here: lightweight annotations with run-time checks and trusted code where necessary.

3.2 Collaboration

A consequence of applying our tools to the Linux kernel is that we have generated a large amount of information about functions and types in the Linux kernel in a form that is usable by the compiler. Some of this information was generated manually by reading comments and code, while other properties were inferred by our tools. In order to make this information available to other researchers and programmers, we propose the creation of a *collaborative database* of source code information that would allow different researchers and tools to share and reuse information about publicly available source code such as the Linux kernel.

For example, this database could provide pointer alias information and bounds information for function arguments and global variables within Linux. This information is required by both Deputy and CCount, and it will almost certainly be of use to future analyses. We can also store information about blocking functions, error codes, and so on. In addition to aiding researchers, this information would also provide a useful reference for programmers who wish to see additional invariants that are not specified directly in the code or comments. Indeed, with the wide variety of possible analyses that we propose, it may be useful for the programmer to store this information on the side instead of cluttering up the code directly. In addition, the ability to incorporate this information from an external source may be useful when trying to maintain this information across multiple versions of the kernel.

We have seeded this repository of annotations at our web page: `http://ivy.cs.berkeley.edu/`. Our Linux annotations are available here, and we encourage other researchers to help us in expanding the scope of these annotations.

4 Related Work

We previously described the Deputy type system [CHA+07], and the application of Deputy to Linux device drivers in SafeDrive [ZCA+06]. Some of us worked on on CCured [NCH+05], a predecessor to Deputy. Here, we present our experience applying Deputy to a complete, bootable Linux kernel, and discuss the basic principles of both Deputy and our other tools that allow us to scale these analyses to large programs.

CSSV [DRS03], Saber-C [KLP88], and a number of other projects [ABS94, SV98] are capable of verifying some soundness properties for C programs. However, we are not aware of any previous attempt to apply such tools to a program as large and complex as the Linux kernel.

In addition, there exist several safe C variants, such as CCured [NCH+05] and Cyclone [JMG+02], which attempt to impose a stricter typing discipline on C programs. However, these systems require changes to data structures that make an incremental transition to these C variants difficult.

Eau Claire [Che02], MC [HCXE02], and MECA [YKXE03] are three examples of bug-finding tools for systems software. While these tools find many important bugs, they do not guarantee that no more bugs exist, and they do not prevent bug reintroduction. Also, each run of these tools requires a programmer to sift through false positives every time the tool is used, whereas our approach yields a modified program that checks cleanly after the initial programmer effort.

Projects such as Melange [MHD+07], JavaOS [Mit96], and Inferno [WP97] have attempted to write systems code in safe languages. However, when legacy code already exists in C, we believe it would be easier to apply our soundness tools to this legacy code than to rewrite the code in a safe language. Our approach focuses on incremental tools that allow programmers to preserve their investment in existing code while improving its reliability.

5 Conclusion

It is estimated that software vulnerabilities cost $13 billion in 2001, $30 billion in 2002, and $55 billion in 2003 [New04]. While bug-finding tools can be very helpful in finding some of these defects, sound static analyses allow us to guarantee their absence.

In this paper, we have discussed our experience thus far in applying soundness tools to the Linux kernel. Our results are encouraging: we were able to rule out many type errors and buffer overruns in 435,000 lines of kernel code with only 7 weeks of effort, and we were able to verify 98% of the deallocations in this code with only 6 weeks of effort. We thus have reason to believe that it is both practical and wise to focus on making systems software completely safe against such defects.

Notes

[1]So far we have only modified `kmalloc`, `kfree`, and the slab allocators, but extending this support to `vmalloc`, `vfree`, and `alloc_page` should be straightforward.

[2]At the time of writing, the kernel version of CCount does not track references from local variables; however, we expect to have this feature implemented soon.

[3]We also null out the pointer passed to free functions.

References

[ABS94] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, 1994.

[BCZ+03] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP*, 2003.

[BS97] A. Brown and M. Seltzer. Operating system benchmarking in the wake of Lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *SIGMETRICS*, 1997.

[CHA+07] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula. Dependent types for low lovel programming. In *European Symposium on Programming*, 2007.

[Che02] B. Chess. Improving computer security using extended static checking. In *IEEE Security and Privacy*, 2002.

[DRS03] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.

[HCXE02] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, 2002.

[HLA+05] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[JMG+02] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[KLP88] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: an interpreter-based programming environment for the C language. In *USENIX Summer Conference*, 1988.

[MHD+07] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: Creating a "functional" internet. In *EuroSys*, 2007.

[Mit96] J. Mitchell. JavaOS: Back to the future (abstract). In *OSDI*, 1996.

[NCH+05] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[New04] ZDNet News. PC viruses spawn $55 billion loss in 2003, Jan 2004.

[SV98] G. Smith and D. Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1–3):49–72, 1998.

[WP97] P. Winterbottom and R. Pike. The design of the Inferno virtual machine. In *IEEE Compcon*, 1997.

[YKXE03] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *ACM Computer and Communications Security*, 2003.

[ZCA+06] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *;login:*, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see http://www.usenix.org/membership/specialdisc.html

For more information about membership, conferences, or publications, see http://www.usenix.org.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at http://www.sage.org.

Thanks to USENIX & SAGE Supporting Members

Ajava Systems, Inc.

Cambridge Computer Services, Inc.

cPacket Networks

DigiCert® SSL Certification

EAGLE Software, Inc.

FOTO SEARCH Stock Footage and Stock Photography

Google

GroundWork Open Source Solutions

Hewlett-Packard

IBM

Infosys

Intel

Interhack

MSB Associates

NetApp

Oracle

Raytheon

Ripe NCC

rTIN Aps

Sendmail, Inc.

Splunk

Sun Microsystems, Inc.

Taos

Tellme Networks

UUNET Technologies, Inc.

VMware

Zenoss